Introduction to R: making visuals and exploratory data analysis

Erin Shellman

April 4, 2016

Contents

Preparation **1**

Strap in! . . . 1

R Basics . . . 2

Basic calculation . . . 2

Test your knowledge . . . 3

Data structures . . . 3

Reading in data . . . 5

Slicing and dicing . . . 7

Installing and loading packages . . . 9

Getting help . . . 9

Graphics . . . 10

Introduction to *ggplot2* . . . 11

Aesthetics . . . 15

Factors . . . 16

Faceting . . . 16

Assignment 1 . . . 18

Exploratory data analysis . . . 19

Capital Bikeshare data . . . 19

Formulating exploratory questions . . . 20

Advanced visualizations with *ggpairs* . . . 21

Assignment 2 . . . 22

Aggregation and summarization with *dplyr* . . . 22

Assignment 3 . . . 22

Preparation

Strap in!

Exploratory data analysis (EDA) is the second step of the data mining process, and it's a perfect way to get acquainted with our tools. This notebook is a guide to help you get started with R. It's inevitably incomplete,

so get ready to Google! For those who are new to R, new to working with data, new to programming, or all three please keep in mind:

> Whenever you're learning a new tool, for a long time you're going to suck... But the good news is that is typical, that's something that happens to everyone, and it's only temporary.

That's sage advice from Hadley Wickham, a statistician and software developer who has revolutionized the way that people analyze data with R. We're going to use many of his packages in this class, and you'll be a pro in no time.

R Basics

Basic calculation

You can use R just like a calculator. It's so intuitive I probably don't have to tell you, but I will anyway! Here's all the arithmetic operators you can use:

Operator	Behavior
+	Add scalar values or vectors
-	Subtract scalar values or vectors
	Multiply scalar values or vectors
/	Divide scalar values or vectors
^	exponentiate scalar values or vectors
%%	modulo on scalar values or vectors
%/%	integer division on scalar values or vectors

You can type things like this right into R:

```
23 + 45
```

```
## [1] 68
```

```
value = (4.59 / 0.1)^3
print(value)
```

```
## [1] 96702.58
```

```
# the coolest part is that R will do these operations element-wise on vectors
vector1 = c(1, 2, 3, 4)
vector2 = c(2, 2, 2, 2) # 'c' is short for Concatenate

vector1 + vector2
```

```
## [1] 3 4 5 6
```

```
vector1 * vector2
```

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

```
vector1^2
```

```
## [1]  1  4  9 16
```

Test your knowledge

1. If `x = 3` and `y = 8` what is x^y?
2.

Data structures

R has four primary data structures:

- vectors (or arrays)
- matrices
- data frames
- lists

Vectors

Vectors are one-dimensional structures that contain data of the same type. For example `age = c(18, 30, 27, 59)` is a *numeric* vector and `relationship = c('sister', 'aunt', 'nephew')` is a vector of strings. We can convert the *relationship* vector is a special categorical type called a *factor* like this, `relationship = factor(c('sister', 'aunt', 'nephew'))`. Factors are essential for plotting, and we'll talk about them in more detail later on.

Matrices

Matrices are just like vectors in two-dimensions. They are defined and accessed like this:

```
my_matrix = matrix(data = c(1, 2, 3, 4, 5, 6), ncol = 3)
my_matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
# say we want to grab the even numbers.
# we can index into my_matrix like this:
my_matrix[2, ]
```

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

```
# but what if they don't happen to all be in the second row?
my_matrix[(my_matrix %% 2) == 0]
```

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

```
# how'd I do that?!
```

Data frames

We're going to focus on data frames for most of this class. Data frames are like spreadsheets, they can have column names (*headers*) and can contain data of different types, like factors and numerical values. Headers are great because we can reference columns by name!

```
# collect two vectors in a data frame
df = data.frame(age = c(10, 20, 45, 37),
                relationship = c('sister', 'cousin', 'father', 'aunt'))
df
```

```
##   age relationship
## 1  10       sister
## 2  20       cousin
## 3  45       father
## 4  37         aunt
```

```
# use '$' to select a column by name
df$relationship
```

```
## [1] sister cousin father aunt
## Levels: aunt cousin father sister
```

```
# 'mean' is a built-in R function
mean(df$age)
```

```
## [1] 28
```

```
# or
mean(df[ , 'age'])
```

```
## [1] 28
```

Lists

We won't use lists much in this class, but they're pretty useful. Lists are collections of objects indexed by a key. The stuff inside of a list doesn't need to be the same dimensions or type, and that makes lists a convenient data structure for storing collections of related items:

```
my_list = list(value = 1, array = c(1,2), another_array = 1:5)
my_list
```

```
## $value
## [1] 1
##
## $array
## [1] 1 2
##
## $another_array
## [1] 1 2 3 4 5
```

```
# get the list keys
names(my_list)
```

```
## [1] "value"         "array"         "another_array"
```

```
my_list$array
```

```
## [1] 1 2
```

I use lists most often when I'm computing something in batches and want to store the output in an intermediate object before, say, combining all the results. For example, maybe I'm fitting a monthly forecast and I have three years of data. I could write a function that does something like this:

```
while I still have data left:
  loop through data month-by-month:
    fit model
    results[month] = fitted model
    return(results)
```

Then, at the end of this code I've got a list, indexed by month, that contains all my models and their associated attributes.

Reading in data

There're lots of ways to read data into R, but in this class you can get away with always copying/pasting this:

```
# set your working directory - normally where you data are
setwd('path/to/your/data')

data = read.delim('data.file',
                  header = TRUE,
                  sep = '\t')
```

Type `?read.delim` to learn what the `header` and `sep` arguments do.

To get acquainted, we'll use a data set called *diamonds*, that comes with the *ggplot2* package. It's a data set containing physical characteristics and prices of about 54,000 diamonds. Loading pre-loaded data in R is simple:

Once data are loaded, there are tons of ways to spot-check it:

```
help(diamonds)
data(diamonds)
```

Once data are loaded, there are tons of ways to spot-check it:

```
dim(diamonds) # print dimensions
```

```
## [1] 53940    10
```

```r
names(diamonds) # print column names
```

```
##  [1] "carat"   "cut"     "color"   "clarity" "depth"   "table"   "price"
##  [8] "x"       "y"       "z"
```

```r
str(diamonds) # data STRucture
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    53940 obs. of  10 variables:
##  $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 0.23 ...
##  $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 3 3 1 3 ...
##  $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 6 5 2 5 ...
##  $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2 6 7 3 4 5 ...
##  $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
##  $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
##  $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
##  $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
##  $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
##  $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...
```

```r
head(diamonds, 3) # print top 3 rows
```

```
## Source: local data frame [3 x 10]
##
##   carat     cut  color clarity depth table price     x     y     z
##   (dbl)  (fctr) (fctr)  (fctr) (dbl) (dbl) (int) (dbl) (dbl) (dbl)
## 1  0.23   Ideal      E     SI2  61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium      E     SI1  59.8    61   326  3.89  3.84  2.31
## 3  0.23    Good      E     VS1  56.9    65   327  4.05  4.07  2.31
```

```r
tail(diamonds, 3) # print bottom 3 rows
```

```
## Source: local data frame [3 x 10]
##
##   carat       cut  color clarity depth table price     x     y     z
##   (dbl)    (fctr) (fctr)  (fctr) (dbl) (dbl) (int) (dbl) (dbl) (dbl)
## 1  0.70 Very Good      D     SI1  62.8    60  2757  5.66  5.68  3.56
## 2  0.86   Premium      H     SI2  61.0    58  2757  6.15  6.12  3.74
## 3  0.75     Ideal      D     SI2  62.2    55  2757  5.83  5.87  3.64
```

```r
summary(diamonds) # summarize the columns
```

```
##      carat               cut        color        clarity
##  Min.   :0.2000   Fair     : 1610   D: 6775   SI1    :13065
##  1st Qu.:0.4000   Good     : 4906   E: 9797   VS2    :12258
##  Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194
##  Mean   :0.7979   Premium  :13791   G:11292   VS1    : 8171
##  3rd Qu.:1.0400   Ideal    :21551   H: 8304   VVS2   : 5066
##  Max.   :5.0100                     I: 5422   VVS1   : 3655
##                                     J: 2808   (Other): 2531
##      depth           table           price             x
```

```
##  Min.   :43.00   Min.   :43.00   Min.   :  326   Min.   : 0.000
##  1st Qu.:61.00   1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710
##  Median :61.80   Median :57.00   Median : 2401   Median : 5.700
##  Mean   :61.75   Mean   :57.46   Mean   : 3933   Mean   : 5.731
##  3rd Qu.:62.50   3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540
##  Max.   :79.00   Max.   :95.00   Max.   :18823   Max.   :10.740
##
##        y               z
##  Min.   : 0.000   Min.   : 0.000
##  1st Qu.: 4.720   1st Qu.: 2.910
##  Median : 5.710   Median : 3.530
##  Mean   : 5.735   Mean   : 3.539
##  3rd Qu.: 6.540   3rd Qu.: 4.040
##  Max.   :58.900   Max.   :31.800
##
```

Slicing and dicing

Matrices and data frames are indexed `df[row, col]`. Leaving either the `row` or `col` element blank tells R to take all rows, or all columns. There are many ways to slice up data frame in R for example:

```
# 1. grab a column by name and assign to edu
price = diamonds[15:20, 'price']
print(price)
```

```
## Source: local data frame [6 x 1]
##
##   price
##   (int)
## 1   345
## 2   345
## 3   348
## 4   351
## 5   351
## 6   351
```

```
# 2. or use the $ to grab the whole column
price = diamonds$price
head(price)
```

```
## [1] 326 326 327 334 335 336
```

```
# 3. we can even combine the two to get rows 15 - 20 of the Education column
price = diamonds[15:20, ]$price
print(price)
```

```
## [1] 345 345 348 351 351 351
```

```
# 4. what if we don't know the row numbers, but we have a condition?
pricey = subset(diamonds, price > 18800)
print(pricey)
```

```
## Source: local data frame [5 x 10]
##
##   carat       cut  color clarity depth table price     x     y     z
##   (dbl)    (fctr) (fctr)  (fctr) (dbl) (dbl) (int) (dbl) (dbl) (dbl)
## 1  2.00 Very Good      H     SI1  62.8    57 18803  7.95  8.00  5.01
## 2  2.07     Ideal      G     SI2  62.5    55 18804  8.20  8.13  5.11
## 3  1.51     Ideal      G      IF  61.7    55 18806  7.37  7.41  4.56
## 4  2.00 Very Good      G     SI1  63.5    56 18818  7.90  7.97  5.04
## 5  2.29   Premium      I     VS2  60.8    60 18823  8.50  8.47  5.16
```

```
# 5. select just some of the columns with dplyr's 'select' function
library(dplyr)
sub = select(diamonds, carat, cut, price)
head(sub)
```

```
## Source: local data frame [6 x 3]
##
##   carat       cut price
##   (dbl)    (fctr) (int)
## 1  0.23     Ideal   326
## 2  0.21   Premium   326
## 3  0.23      Good   327
## 4  0.29   Premium   334
## 5  0.31      Good   335
## 6  0.24 Very Good   336
```

```
# 6. and then we can use dplyr's 'filter' to do the same subsetting we did in 3
filtered = filter(sub, carat > 0.40)
head(filtered)
```

```
## Source: local data frame [6 x 3]
##
##   carat       cut price
##   (dbl)    (fctr) (int)
## 1  0.42   Premium   552
## 2  0.70     Ideal  2757
## 3  0.86      Fair  2757
## 4  0.70     Ideal  2757
## 5  0.71 Very Good  2759
## 6  0.78 Very Good  2759
```

```
# 7. what's cool about dplyr is that you can nest those
head(
  select(
    filter(diamonds, carat > 0.40),
  carat, cut, price)
)
```

```
## Source: local data frame [6 x 3]
##
##   carat       cut price
##   (dbl)    (fctr) (int)
```

```
## 1  0.42   Premium   552
## 2  0.70     Ideal  2757
## 3  0.86      Fair  2757
## 4  0.70     Ideal  2757
## 5  0.71 Very Good  2759
## 6  0.78 Very Good  2759
```

```
# 8. or, even cooler, we can use the 'pipe' notation
filter_by_pipe = diamonds %>% filter(carat > 0.40) %>% select(carat, cut, price)
head(filter_by_pipe)
```

```
## Source: local data frame [6 x 3]
##
##   carat       cut price
##   (dbl)    (fctr) (int)
## 1  0.42   Premium   552
## 2  0.70     Ideal  2757
## 3  0.86      Fair  2757
## 4  0.70     Ideal  2757
## 5  0.71 Very Good  2759
## 6  0.78 Very Good  2759
```

Self-check

1. How many diamonds cost less than $500?
2. How many diamonds cost at least $15,000?
3. What is the range of prices for diamonds of color D?

Installing and loading packages

One of the greatest strengths of R is the active community that creates powerful tools and releases them publicly as R *packages*. Today we'll learn about two powerful packages, *ggplot2* and *dplyr*. We'll use *ggplot2* to elegantly create rich visualizations and *dplyr* to quickly aggregate and summarize data.

Packages are easy to install and load:

```
# install
install.packages('dplyr', dependencies = TRUE)
install.packages('ggplot2', dependencies = TRUE)
install.packages('GGally', dependencies = TRUE)
install.packages('scales', dependencies = TRUE)
install.packages('lubridate', dependencies = TRUE)

# load
library(dplyr)
library(ggplot2)
```

Getting help

R docs

You can access documentation for any function in R by typing `help(function_name)` or `?(function_name)`. The help files conform to a standard format so that they're easy to navigate. You'll probably end up reading the *usage*, *arguments*, *value* and *examples* sections most often. The *usage* section describes how to call the function, *arguments* lists all the values that can be set in the call, *value* describes what information the function call will return to you, and *examples* are typically self-contained chunks of code that you can copy and paste into the console.

The Internets

Sometimes it's faster and easier to get help with Google. The first step of my troubleshooting workflow is typing questions literally into Google, with a few specifics about the language of the solution I'm looking for. For example, "R ggplot2 how to change legend font size" returns the following top 3 links:

1. Cookbook for R » Legends (ggplot2)
2. r - increase legend font size ggplot2 - Stack Overflow
3. theme. ggplot2 0.9.2.1

all of which answer the question. The first link is worth bookmarking. It's a page of plots with associated code so you can easily find an example that looks like what you want. The second is another phenomenal resource called Stackoverflow, which is a public forum for asking programming questions and getting answers from the public. I use this website probably over 50 times a day, and I bet you will too. The last link is the *ggplot2* documentation that lists all the arguments in the `theme()` element, one of which is `legend.text`. If you're new to R (and even if you're not) you're going to have to look a lot of things up. If you're stuck, don't spin your wheels, just start typing the problem into Google and you might be surprised at how easy it is to find solutions.

Classmates

Finally, ask your classmates! As a professional data analyst you likely won't be working alone. If you're stuck, chances are that your classmates are too. Even if not, sometimes getting a fresh set of eyes on your code is all you need to find little bugs. Help each other out!

Useful links

- http://www.statmethods.net/
- http://adv-r.had.co.nz/
- http://adv-r.had.co.nz/Style.html
- http://www.cookbook-r.com/
- http://stackoverflow.com/questions/tagged/r-faq%20

Graphics

Visuals are the most compelling way to communicate results. At the exploratory stage, we generate numerous, relatively low-fidelity figures that help familiarize the analyst with new data and guide subsequent analyses. During the exploratory phase, the goal is to generate sensible figures quickly without fretting over details, however **axes should always be labeled**.

Statistician and artist Edward Tufte has canonized some fundamental graphics tips to keep in mind as you create your figures:

- Highlight comparisons
- Show causality

- Show as much as possible. We'll explore faceting, color, shape, size as method to do this.
- Integrate evidence. Where appropriate, include text, numbers, images but only to the extent that they enhance the visualization's narrative.
- Figures **always** have labeled axes!! (ok, this one is mine.)

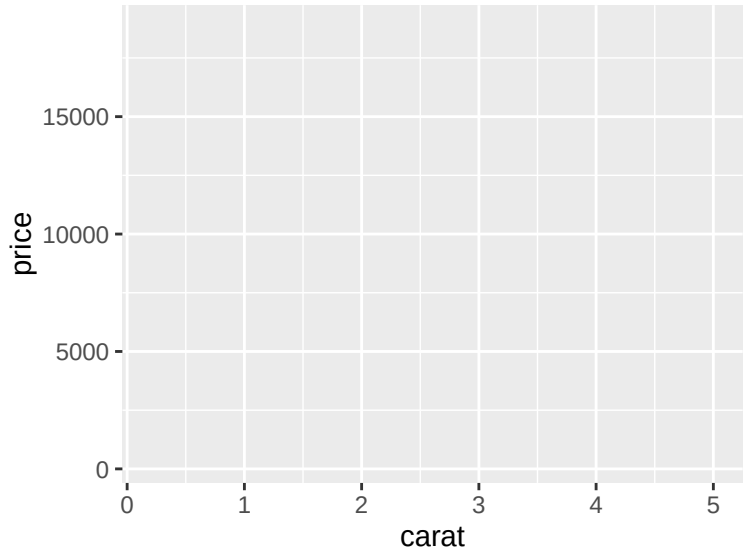
Introduction to *ggplot2*

R has three core plotting systems, *base*, *lattice* and *ggplot2*. In this course we'll use the plotting library *ggplot2* because it combines the best parts of the other two plotting systems and uses a consistent syntax that makes plot code intuitive and reusable.

Plot objects in *ggplot2* are made up of *geoms* and *aesthetics*. Geometric objects, or *geoms*, describe the type of plot, *e.g.* scatter or boxplot. *aesthetics* describe how to draw the plot *e.g.* color, size or location.

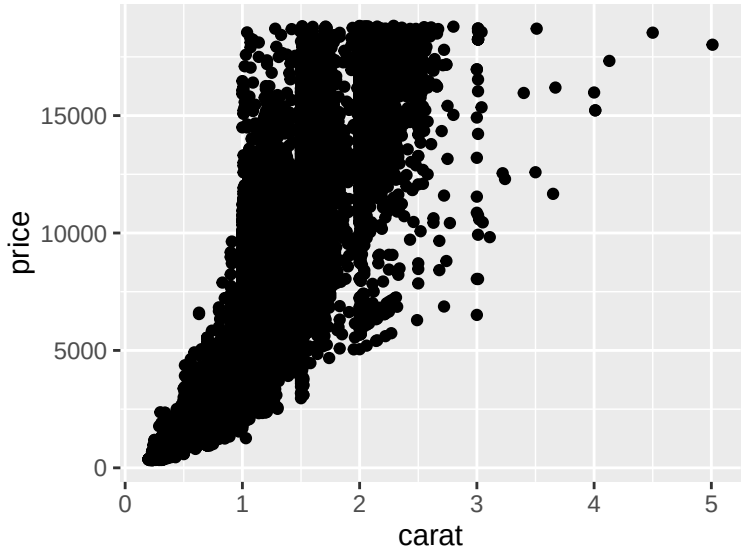
Every ggplot starts with a line like this: `ggplot(dataframe, aes(x = var1, y = var2, ...))` that maps data onto x and y dimensions for our plot, yet to be defined. If you run the aforementioned plotting code what do you get?

```
ggplot(diamonds, aes(x = carat, y = price))
```



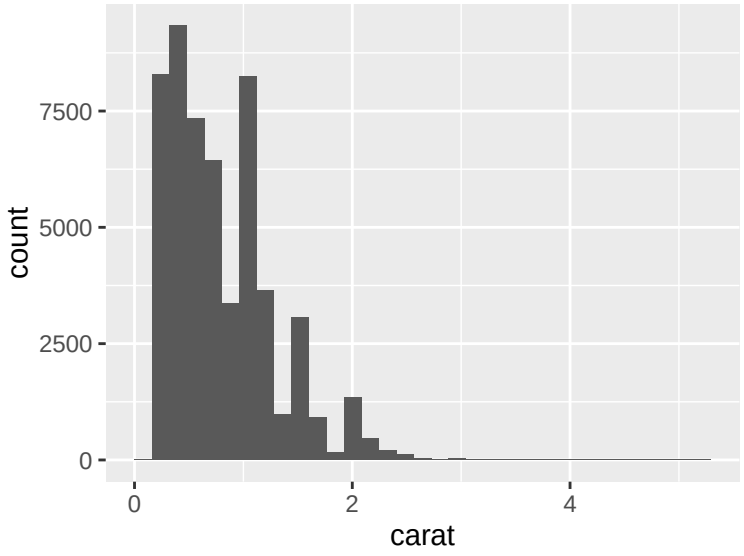
A blank coordinate system with x- and y-axis labels. Why? Well, you've told ggplot2 how to map data to the coordinates x and y, but you haven't told it how to draw that mapping. We tell ggplot how to draw using *geoms*.

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



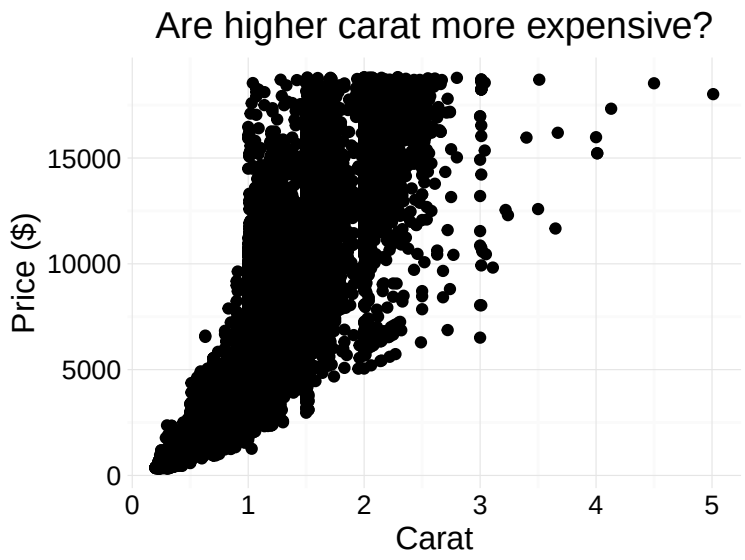
Think of the plots as being built up as layers. First we map the variables with the `ggplot` statement, then we tell ggplot what type of plot to make (e.g. scatter, histogram, bar plot, etc).

```
ggplot(diamonds, aes(x = carat)) + geom_histogram()
```

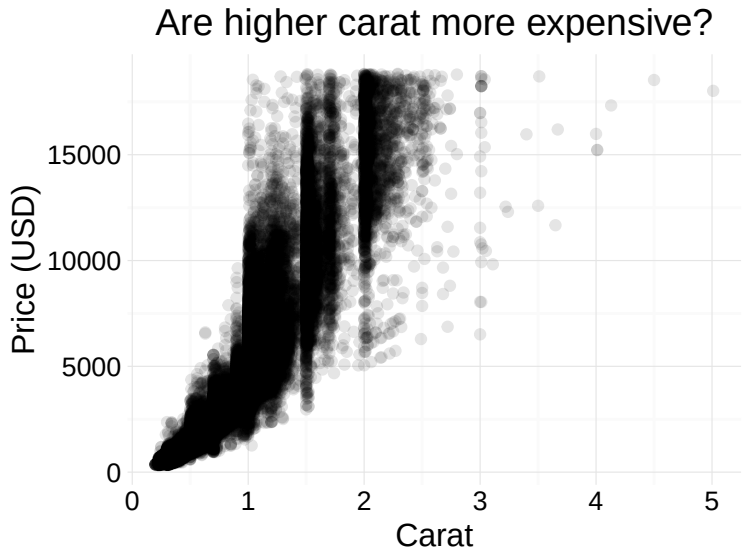


Finally, we can layer on nice labels, colors and annotation. I highly recommend having this page up when plotting: http://docs.ggplot2.org/

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point() +
  scale_x_continuous('Carat') + # axis labels
  scale_y_continuous('Price ($)') +
  ggtitle('Are higher carat more expensive?') +
  theme_minimal() # Themes? Score! How does the plot change w/wo it?
```

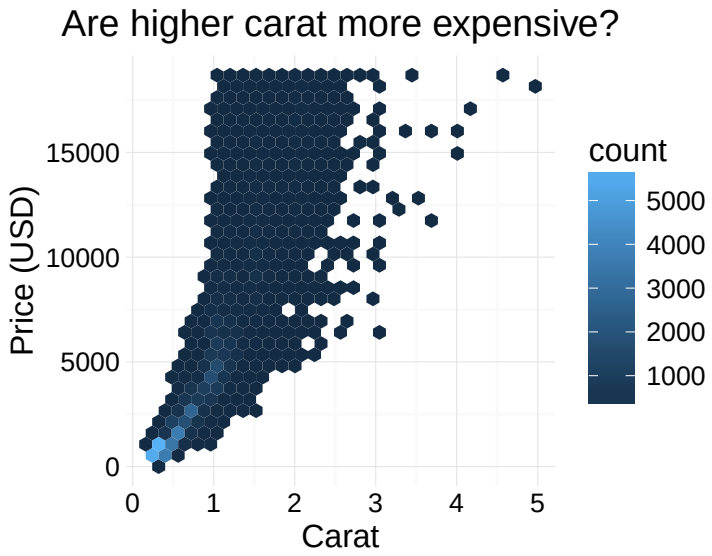



```
# scatter plot
ggplot(diamonds, aes(x = carat, y = price)) +
  # data mapping
  geom_point(alpha = 0.10) + # draw a scatter plot
  # plot labeling
  scale_x_continuous('Carat') +
  scale_y_continuous('Price (USD)') +
  # theme elements
  ggtitle('Are higher carat more expensive?') +
  theme_minimal()
```

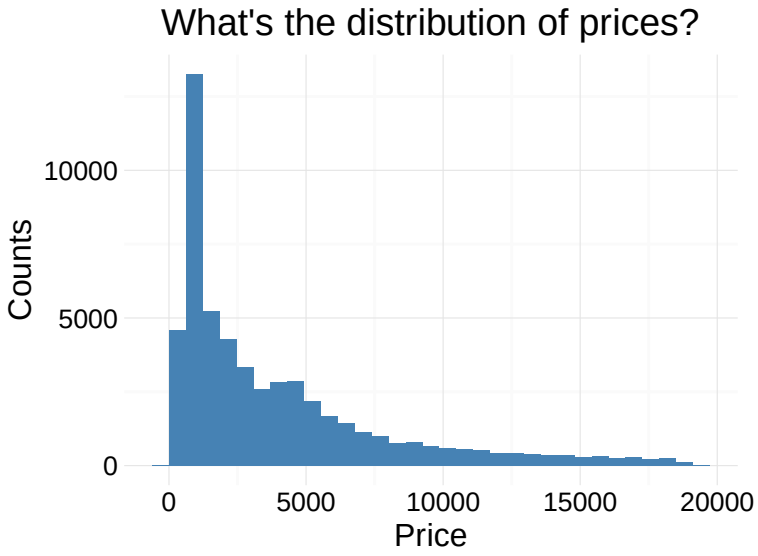



```
# scatter plot
ggplot(diamonds, aes(x = carat, y = price)) +
  # data mapping
  geom_hex() + # draw a scatter plot
  # plot labeling
  scale_x_continuous('Carat') +
```

```
  scale_y_continuous('Price (USD)') +
  # theme elements
  ggtitle('Are higher carat more expensive?') +
  theme_minimal()
```

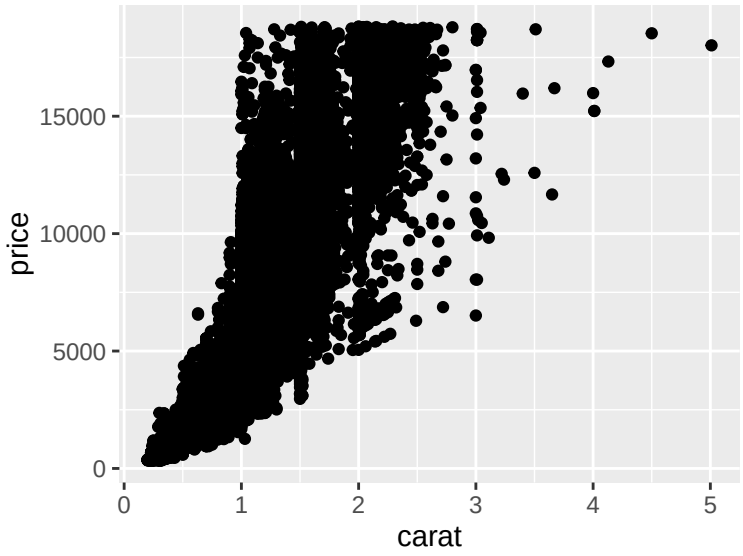



```
# histogram
ggplot(diamonds, aes(x = price)) +
  # data mapping
  geom_histogram(fill = 'steelblue') + #
  # plot labeling
  scale_x_continuous('Price') +
  scale_y_continuous('Counts') +
  # theme elements
  ggtitle("What's the distribution of prices?") +
  theme_minimal()
```



If you're familiar with R's `base` graphics then `qplot()` will seem natural. Feel free to read up on `qplot()` and its uses, but we're going to focus primarily on the `ggplot` function for constructing graphics.

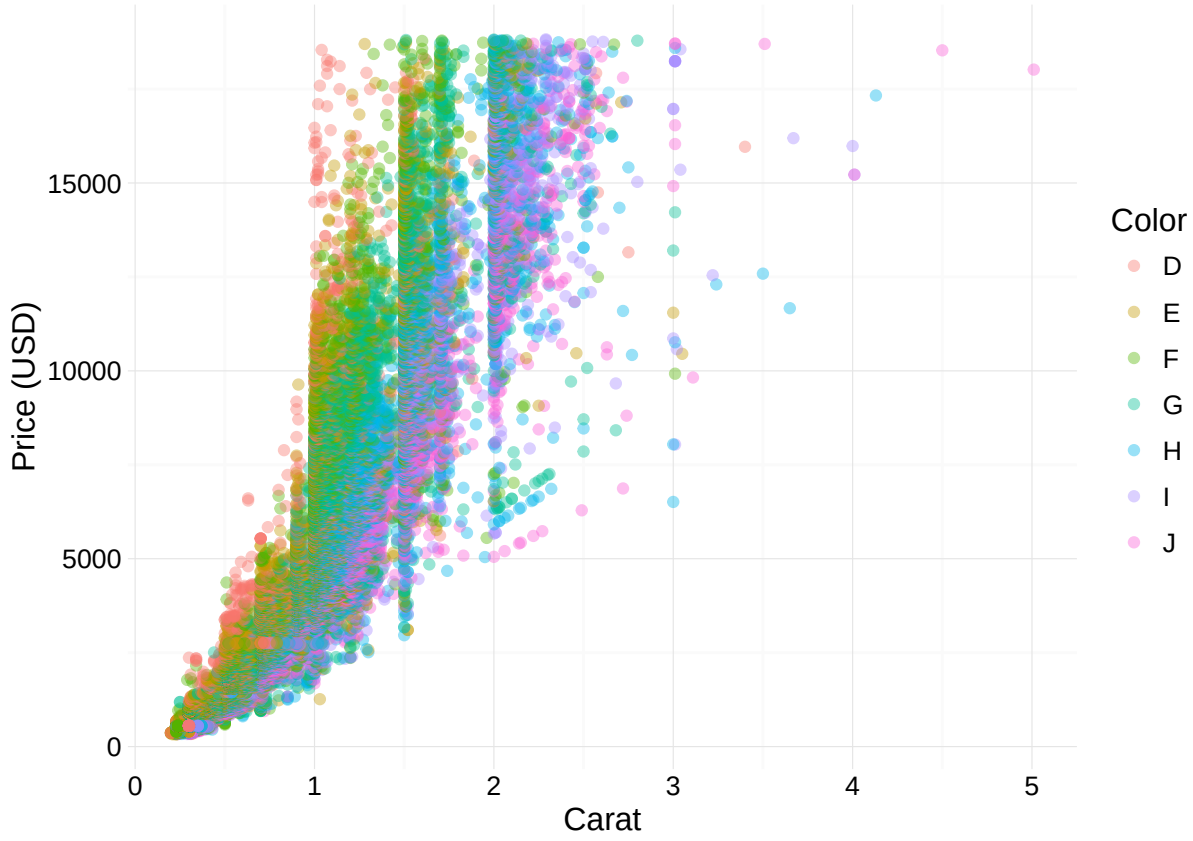
```
# quick scatterplot made with qplot
qplot(carat, price, data = diamonds)
```



Aesthetics

When you want to change a feature of your figure based on the value of another variable, you use aesthetics. Unlike the histogram above, where we could specify the color explicitly in the *geom()*, when the color depends on the value of a variable, it goes inside `aes()`.

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(aes(color = factor(color)), alpha = 0.40) + # draw a scatter plot
  scale_x_continuous('Carat') +
  scale_y_continuous('Price (USD)') +
  scale_color_discrete('Color') + # legend title
  theme_minimal()
```



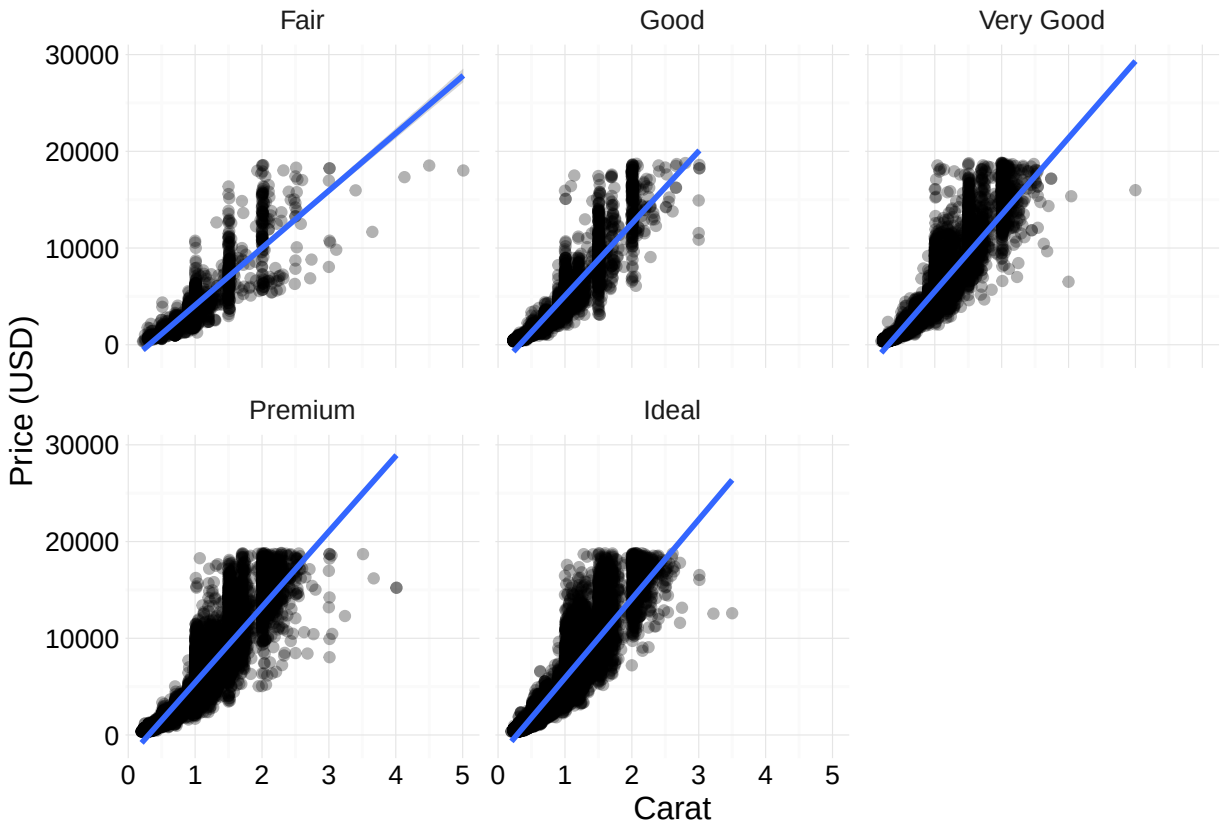
Factors

We used a new function called `factor` in the plot above when describing how to color the points. A `factor` is a variable type in R that represents categorical data, in this case the color of a diamond. Factor variables are very handy in *ggplot2* because they allow us to quickly change color, shape, facet and many other graphical features by the value of a category. Let's try it with facets.

Faceting

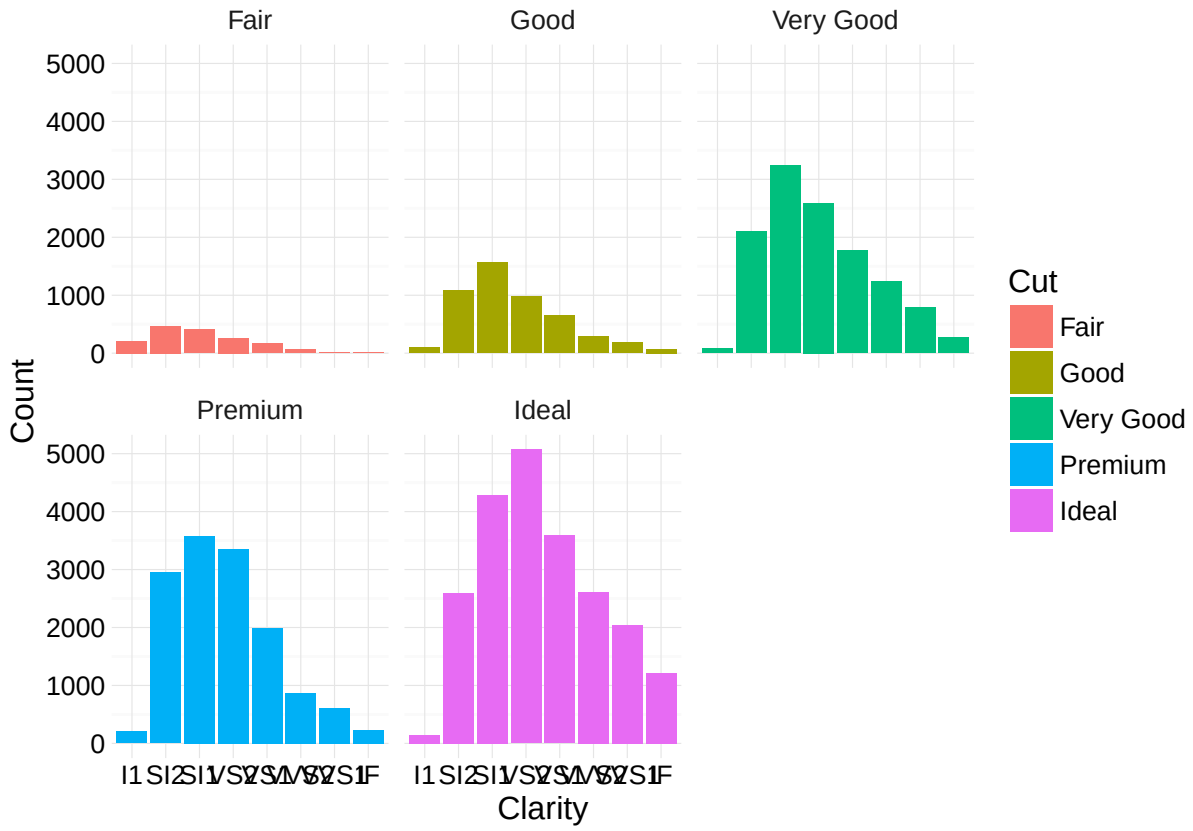
Our colorful carat plot looks pretty good, but there are a lot of points laying on top of each other. When you want to compare plots across groups separately you can use a *facet* to split the figure up by category.

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(alpha = 0.30) +
  geom_smooth(method = 'lm') +
  facet_wrap(~ cut) +
  scale_x_continuous('Carat') +
  scale_y_continuous('Price (USD)') +
  scale_color_discrete('Color') +
  theme_minimal()
```



Now we can view every grouping individually. Notice how we were able to add a smoother to the plot and `facet_wrap` elegantly applied it to each sub-plot. How about with bar plots?

```
ggplot(diamonds, aes(clarity, fill = factor(cut))) +
  geom_bar() +
  facet_wrap(~ cut) +
  scale_x_discrete('Clarity') +
  scale_y_continuous('Count') +
  scale_fill_discrete('Cut') +
  theme_minimal()
```



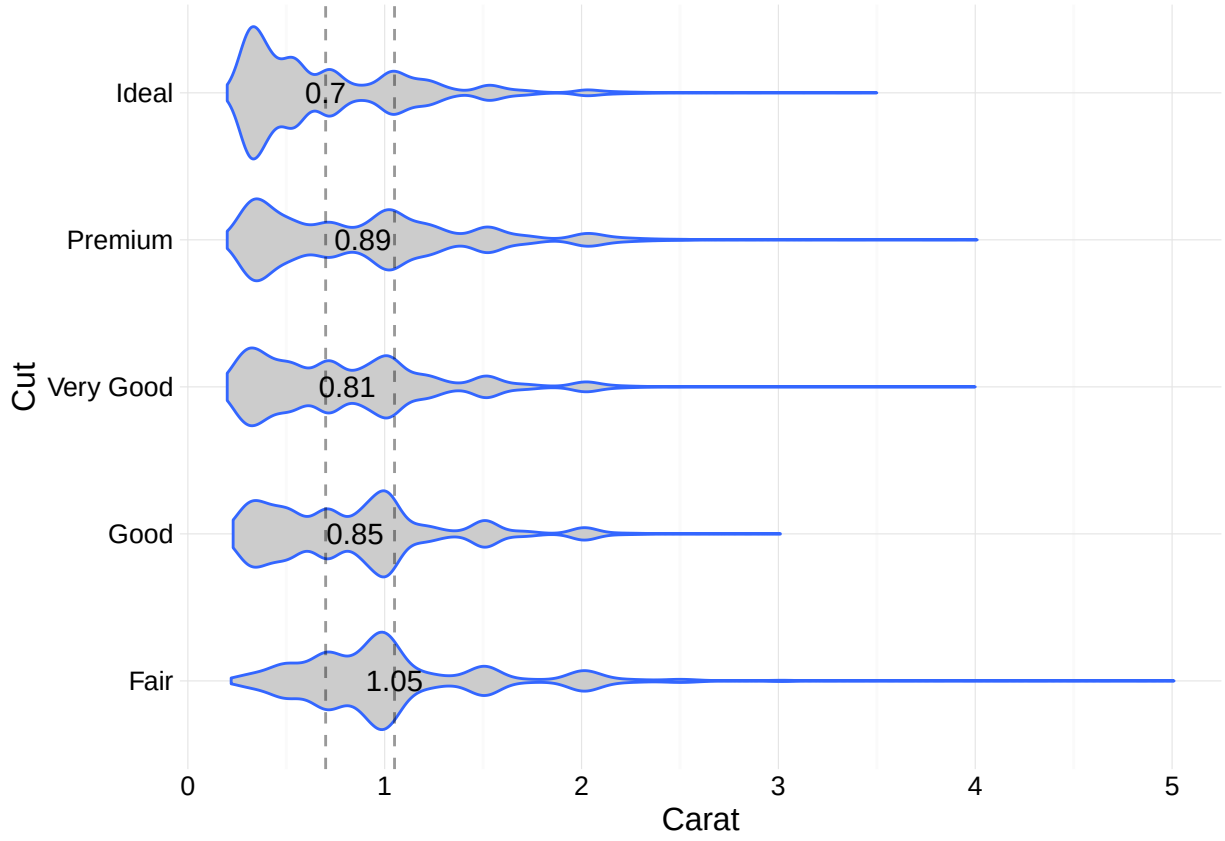
Assignment 1

One or two questions that can be answered with a figure and dplyr.

Annotation on a figure can add clarity and help the audience understand the message. A lot of these features are not necessary for exploratory analyses, but can be very helpful for communicating final results. For example, suppose we're interested in characterizing the joint distributions of carats and cuts. These distributions are bumpy and highly right-skewed, so it's difficult to ascertain the mean without further annotation.

```
carat_summary = diamonds %>% group_by(cut) %>% summarize(carat_mean = round(mean(carat), 2))

ggplot(diamonds, aes(x = factor(cut), y = carat)) +
  geom_violin(fill = "grey80", colour = "#3366FF") +
  geom_text(data = carat_summary, aes(label = carat_mean, x = cut, y = carat_mean)) +
  geom_hline(yintercept = range(carat_summary$carat_mean), alpha = 0.4, linetype = 2) +
  coord_flip() +
  # plot labeling
  scale_x_discrete('Cut') +
  scale_y_continuous('Carat') +
  scale_fill_discrete('Cut') + # legend title
  # theme elements
  theme_minimal()
```



Ignoring the quick aggregation we did for now, we used `geom_text` to write the mean carat of each cut inside of the violin plot. Also we drew the range of those means using `geom_hline` which lets us quickly assess the spread in means. We'll talk about that cool aggregation step in detail in the next section.

Self-check

1. Spend some time exploring the ggplot2 docs and try out some new figures. What did you learn from your figure?

Exploratory data analysis

Capital Bikeshare data

The *diamonds* data was a good warm-up, but let's dive into the data you'll be working with in the first project. The data are from Washington D.C.'s bikeshare program in 2015 and are in the file *bikeshare_2015.tsv* in the course repository.

```
data = read.delim('/your/dir/bikeshare_2015.tsv',
                  header = TRUE,
                  sep = '\t')
```

Read the data in using the `read.delim` function previously described and type `head(data)` to inspect the data set. There's a station name, rental duration, a count of the number of rentals, lat/long, and lots of information about the surrounding environment. Let's start exploring and see if we can make any preliminary observations.

Formulating exploratory questions

Throughout the course you'll be tackling business problems with data which means you'll need to:

1. formalize the business problem as a data mining process
2. articulate the solution *i.e.* how will you know when you've answered the question?
3. help make a decision, *i.e.* what should the business do?

Spend some time looking over the data and getting a sense for what types of information are available. When I scan over a data set I create small hypotheses in my mind and think about the data I'd need to disprove them. For example, I see a variable called *traffic_signals*, I wonder if rentals would be lower in areas with more traffic signals because people don't want to bike around cars?

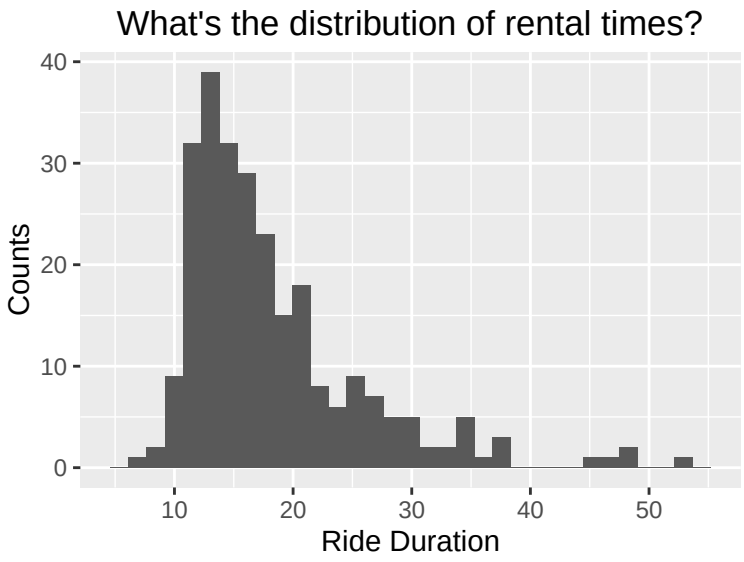
```
# How many stations are there?
length(data$station)
```

```
## [1] 258
```

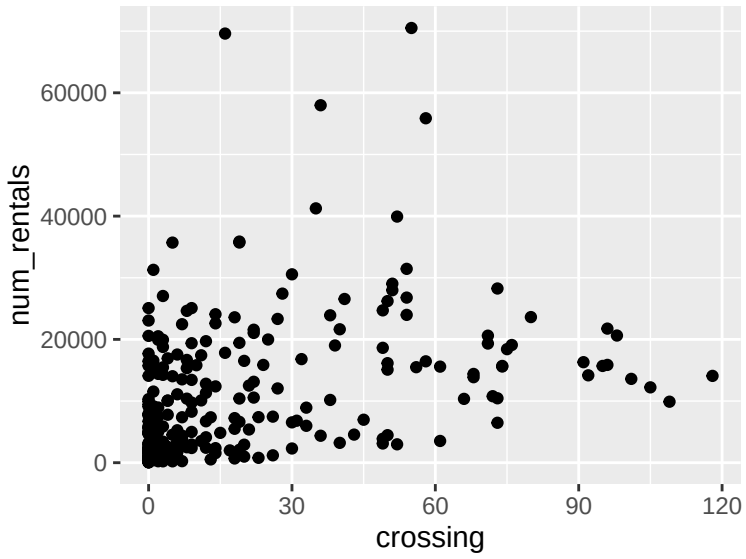
```
# What's the ride duration distribution like?
summary(data$duration_mean)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   6.594  13.000  16.200  18.240  20.930  52.590
```

```
ggplot(data, aes(x = duration_mean)) +
  geom_histogram() + #
  scale_x_continuous('Ride Duration') +
  scale_y_continuous('Counts') +
  ggtitle("What's the distribution of rental times?")
```



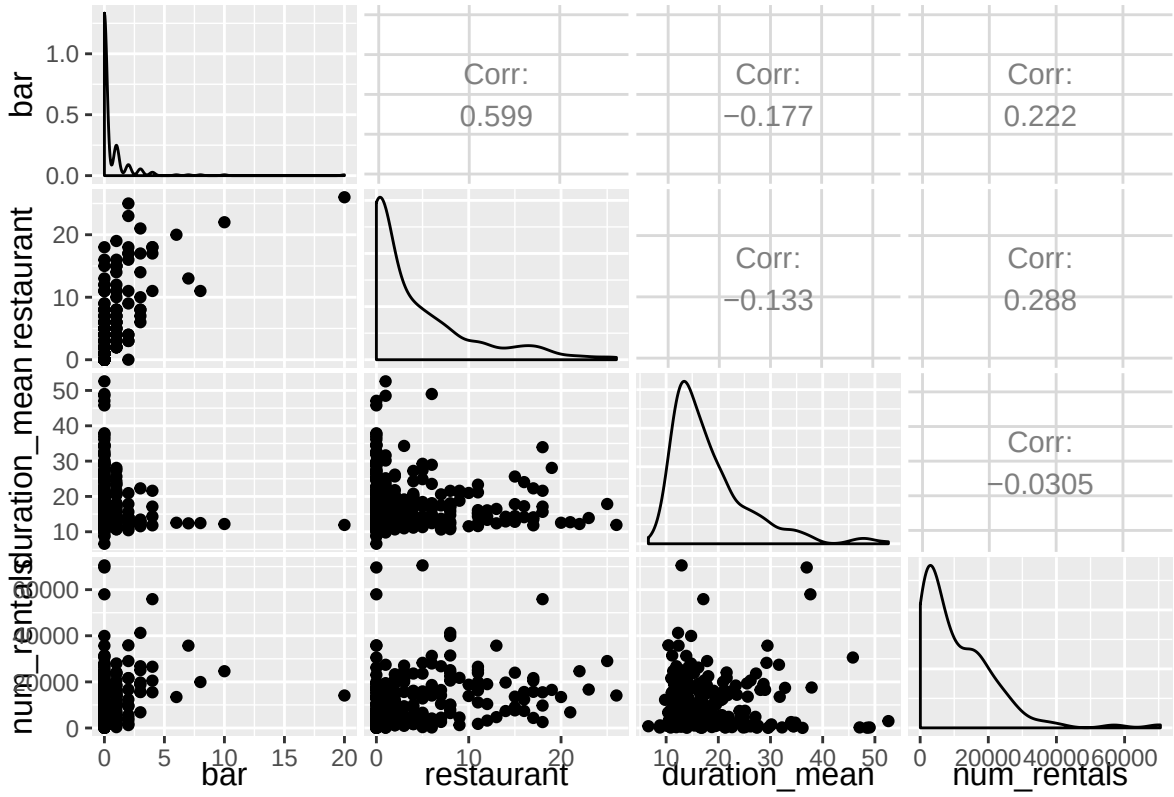

```
# Is there a relationship between the number of crosswalks nearby and rentals?
ggplot(data, aes(x = crossing, y = num_rentals)) +
  geom_point()
```



Advanced visualizations with *ggpairs*

You might have noticed that there are a lot of variables in the data set (if you didn't try running `dim(data)`). It's going to take forever to discover patterns one plot at a time. Fortunately there's a cool function called `ggpairs` in the *GGally* package to plot continuous variables as a matrix.

```
library(GGally) # you might have to install it first
ggpairs(data[ , c('bar', 'restaurant', 'duration_mean', 'num_rentals')])
```



Whoa, what are we looking at here? We have four variables which are shown on both the x- and y-axes. In the lower left corner are scatterplots showing multiple comparisons. For example, the very bottom left-most tile shows the relationship between the number of rentals and the number of nearby bars. The diagonal shows the marginal distribution of each variable, which shows us that all four variables are right-skewed (*i.e.* long tail going towards the right). Finally, the top right corner shows the correlation of each comparison. For example the correlation between the number of rentals and number of nearby bars is 0.22.

Assignment 2

1. We hypothesize that `tourism_artwork` and `railway_level_crossing` are correlated with `duration_mean` and `num_rentals`. Test that hypothesis by using *ggpairs*. What are the correlations?

2. Make some hypotheses about three additional variables that could be positively and negatively correlated with `duration_mean` and `num_rentals`. Test your hypotheses using the plotting and summary methods we've learned so far or by Googling for others.
    - *Self-check*: how do we generate a list of the variable names in the data set?

Aggregation and summarization with *dplyr*

Virtually any analysis will require aggregation techniques to summarize and compress data. Aggregation can also be used to create new variables to explore. The package *dplyr* is tied for my personal favorite package in R (with *ggplot2*) and it is an extremely powerful and elegant way to aggregate, manipulate, tranform, and generally do awesome stuff with data. We saw a preview of *dplyr* in the violin plot, but that was really just scratching the surface of it's capabilities.

dplyr is a fast, consistent tool for working with data frame like objects, and allows you to 'chain' together SQL-like verbs to quickly and easily manipulate data.

Here's a list of all the *dplyr* verbs:

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

The verbs are chained together either by nesting, or by using 'pipes.' The *dplyr* pipe (originally from a package called *magrittr*), allows you to string operations together without saving the intermediate outputs. It is an extremely powerful method of aggregating and summarizing data.

Clearly some stations are more successful than others. It's your job in project 1 to determine if station success can be predicted by these data.

Assignment 3

1. Use combinations of the *dplyr* verbs to familiarize yourself with the data. Are you able to identify any interesting patterns?

2. Need more specific questions.

** Save your code! **