Introduction to R: making visuals and exploratory data analysis

April 4, 2016

Contents

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 'c'oncatenate

vector1 + vector2
```

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

```
vector1 * vector2
```

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

```
vector1^2
```

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

Self-check

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

Data types

Most data types in R are some variant of numerical or categorical. You can find out the type of anything by using the function `is()`, which takes virtually everything as an input and returns its type as the output.

Numeric

R has your standard numerical types that include integers and floating point numbers. Typically there are few practical difference between integers and floats (or doubles), but it all depends on your application.

```
int = 3
is(int)
```

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

```
float = 3.0
is(float)
```

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

```
int == float
```

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

Categorical

Categorical data types get a little more interesting. We'll primarily use two types, logicals and factors.

Logical

Logical or Boolean types are a special type of categorical variable that can only take on two values, true or false.

```
# R is case sensitive, and expects all caps for booleans
x = TRUE
is(x)
```

```
## [1] "logical" "vector"
```

```
y = FALSE

# what is the type returned by is when the input is this assertion?
is(x == y)
```

```
## [1] "logical" "vector"
```

Factors

Factors are special categorical variables in R. They are simultaneoulsy very powerful and very, very annoying to use. Factors are used to categorize data in an array and store it as levels.

```
season = c('Summer', 'Summer', 'Winter', 'Spring', 'Autumn', 'Winter')

# not a factor yet!
is(season)
```

```
## [1] "character"           "vector"              "data.frameRowLabels"
## [4] "SuperClassMethod"
```

```
# apply the factor function and specify the level ordering
season = factor(season, levels = c('Winter', 'Spring', 'Summer', 'Autumn'))
season
```

```
## [1] Summer Summer Winter Spring Autumn Winter
## Levels: Winter Spring Summer Autumn
```

Data structures

R has four primary data structures:

1. vectors (or arrays)
2. matrices
3. lists
4. data frames

Vectors

Vectors (or arrays) 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. You can think or vectors as being similar to a single column in a spreadsheet.

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?!
```

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(single_value = 1, text_array = c('oh hey!', "what's up?"), integer_array = 1:5)
my_list
```

```
## $single_value
## [1] 1
##
## $text_array
## [1] "oh hey!"    "what's up?"
##
## $integer_array
## [1] 1 2 3 4 5
```

As you can see, lists don't have to contain elements of the same size or even the same type. You can conveniently access each element by name, or index:

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

```
## [1] "single_value"  "text_array"    "integer_array"
```

```
# reference by name
my_list$text_array
```

```
## [1] "oh hey!"    "what's up?"
```

```
# reference by index
my_list[3]
```

```
## $integer_array
## [1] 1 2 3 4 5
```

Referencing list elements by index returns a set of key-value pairs. For example, when we ran `my_list[3]` above, the output included the element name (*i.e.* its key), 'integer_array.' If you want to extract the values so you can do something with them, use two brackets:

```
is(my_list[3]) # oops, it's a list
```

```
## [1] "list"   "vector"
```

```
is(my_list[[3]]) # that's better
```

```
## [1] "integer"             "numeric"             "vector"
## [4] "data.frameRowLabels"
```

```
# use two brackets to access the values
my_list[[1]] + my_list[[3]]
```

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

Data frames

In this class we'll focus almost exclusively on data frames. 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
```

Now that you know how to reference and extract columns from a data frame, you can start passing them into R functions:

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

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

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

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

Functions

I just computed a mean using a built-in function called `mean`. In programming, functions are used to organize computer instructions into a collection that can be referred to by name. If you've got computer instructions that repeat often, like for computing the mean, you can save yourself a ton of time and code by encapsulating them inside a function and *call*ing that function by name every time you want to execute the instructions.

Functions typically take parameters or *arguments* as inputs and then return some kind of output. When using built-in functions, the inputs and outputs are determined by the author of the function, and you should familiarize yourself with the function to avoid unexpected behaviors. We'll talk later about how to find and read documentation about R's built-in functions.

Built-in functions

R comes with a ton of functions built-in and there are many more available to you through a rich collection of external packages (more on those later). We'll be using a tiny fraction of the total available in class so if there's a function you want to use but don't know what it's called, don't be shy about Googling/asking in class.

```
mean(df$age) # just like =AVERAGE() in Excel
```

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

```
sd(df$age)
```

```
## [1] 15.89549
```

```
median(df$age)
```

```
## [1] 28.5
```

```
table(df$relationship)
```

```
##
##   aunt cousin father sister
##      1      1      1      1
```

User-defined functions

Even after Googling and asking around, sometimes you'll need to do something so unique or custom to your application that there's nothing pre-packaged available. In that case you can define your own functions.

```
# make input
my_friends = c('Bryan', 'Gary', 'Wendy', 'Uncle Brian', 'Bernease')

# define function
say_hi = function(names) {
  greetings = c('Hiya', 'Good morning,', 'Hello')

  for (name in names) {
```

```
    print(paste(sample(greetings, 1), name, sep = ' '))
  }
}

# then 'call' the function by name
say_hi(my_friends)
```

```
## [1] "Good morning, Bryan"
## [1] "Good morning, Gary"
## [1] "Hiya Wendy"
## [1] "Good morning, Uncle Brian"
## [1] "Hiya Bernease"
```

Reading in data

From a file

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')

# you could also specify the complete path
data = read.delim('path/to/your/data/data.file',
                  header = TRUE,
                  sep = '\t')
```

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

From a package

R and some R packages also come with example datasets already installed. To load pre-bundled data, use `data()`:

```
help(mtcars)
data(mtcars)
```

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

```
dim(mtcars) # print dimensions
```

```
## [1] 32 11
```

```
names(mtcars) # print column names
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

```
str(mtcars) # data STRucture
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```
head(mtcars, 3) # print top 3 rows
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4     21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710    22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
tail(mtcars, 3) # print bottom 3 rows
```

```
##                mpg cyl disp  hp drat   wt qsec vs am gear carb
## Ferrari Dino  19.7   6  145 175 3.62 2.77 15.5  0  1    5    6
## Maserati Bora 15.0   8  301 335 3.54 3.57 14.6  0  1    5    8
## Volvo 142E    21.4   4  121 109 4.11 2.78 18.6  1  1    4    2
```

```
summary(mtcars) # summarize the columns
```

```
##       mpg             cyl             disp             hp
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0
##       drat             wt             qsec             vs
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000
##        am              gear            carb
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000
```

```
##  Median :0.0000   Median :4.000   Median :2.000
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Slicing and dicing

Matrices and data frames are indexed by row and column, `df[row, col]`. Leaving either the `row` or `col` index blank tells R to select 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 a new variable
mpg = mtcars[15:20, 'mpg']
print(mpg)
```

```
## [1] 10.4 10.4 14.7 32.4 30.4 33.9
```

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

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1
```

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

```
## [1] 10.4 10.4 14.7 32.4 30.4 33.9
```

```
# 4. what if we don't know the row numbers, but we have a condition?
efficient = subset(mtcars, mpg > 30)
print(efficient)
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Fiat 128       32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

```
# 5. select just some of the columns with dplyr's 'select' function
library(dplyr)
sub = select(mtcars, mpg, cyl, wt)
head(sub)
```

```
##                   mpg cyl    wt
## Mazda RX4         21.0   6 2.620
## Mazda RX4 Wag     21.0   6 2.875
## Datsun 710        22.8   4 2.320
## Hornet 4 Drive    21.4   6 3.215
## Hornet Sportabout 18.7   8 3.440
## Valiant           18.1   6 3.460
```

```r
# 6. and then we can use dplyr's 'filter' to do the same subsetting we did in 3
filtered = filter(sub, cyl > 4)
head(filtered)
```

```
##    mpg cyl    wt
## 1 21.0   6 2.620
## 2 21.0   6 2.875
## 3 21.4   6 3.215
## 4 18.7   8 3.440
## 5 18.1   6 3.460
## 6 14.3   8 3.570
```

```r
# 7. what's cool about dplyr is that you can nest those
head(
  select(
    filter(mtcars, cyl > 4),
  mpg, cyl, wt)
)
```

```
##    mpg cyl    wt
## 1 21.0   6 2.620
## 2 21.0   6 2.875
## 3 21.4   6 3.215
## 4 18.7   8 3.440
## 5 18.1   6 3.460
## 6 14.3   8 3.570
```

```r
# 8. or, even cooler, we can use the 'pipe' notation
filter_by_pipe = mtcars %>% filter(cyl > 4) %>% select(mpg, cyl, wt)
head(filter_by_pipe)
```

```
##    mpg cyl    wt
## 1 21.0   6 2.620
## 2 21.0   6 2.875
## 3 21.4   6 3.215
## 4 18.7   8 3.440
## 5 18.1   6 3.460
## 6 14.3   8 3.570
```

Self-check

1. What is the range of MPG?
2. Which car has the most cylinders and best MPG?
3. Which car is the lightest? Which is the heaviest?

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:

```r
# install
install.packages('dplyr', dependencies = TRUE)
install.packages('ggplot2', 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 million 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

Data visualizations 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*

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:

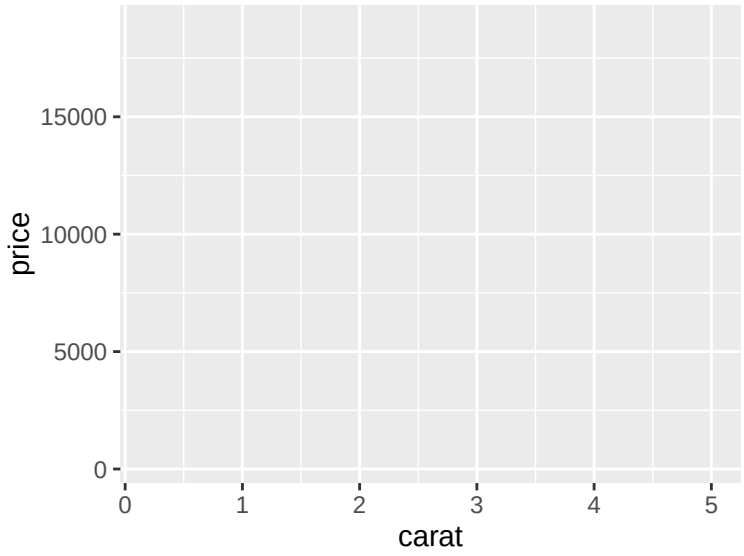
```
help(diamonds)
data(diamonds)
```

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.

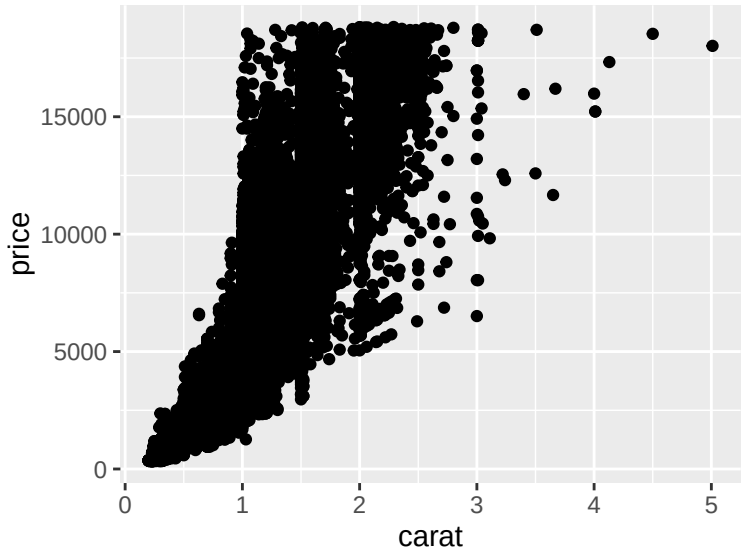
Every ggplot starts with a line like this: `ggplot(dataframe, aes(x = var1, y = var2, ...))` that maps data onto x and y dimensions for our plot, yet to be defined. If you run the aforementioned plotting code what do you get?

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



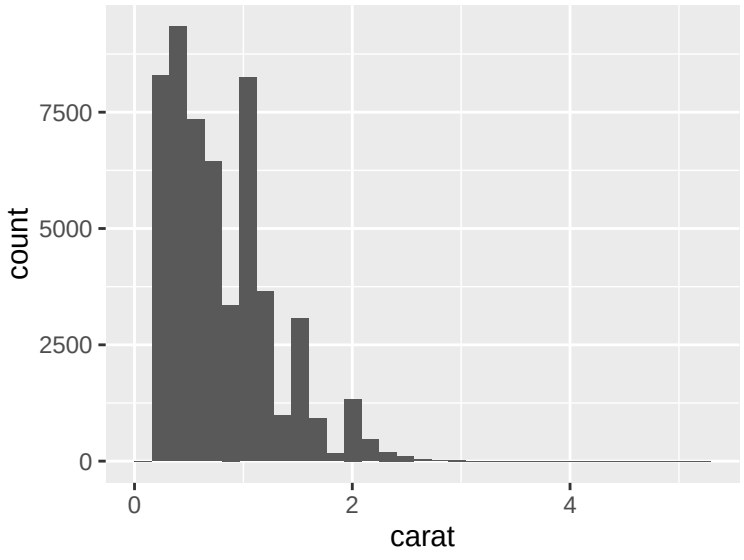
A blank coordinate system with x- and y-axis labels. Why? Well, you told ggplot2 how to map data to the coordinates x and y, but you haven't told it how to draw that mapping. We tell ggplot what to draw by specifiying a *geom*:

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



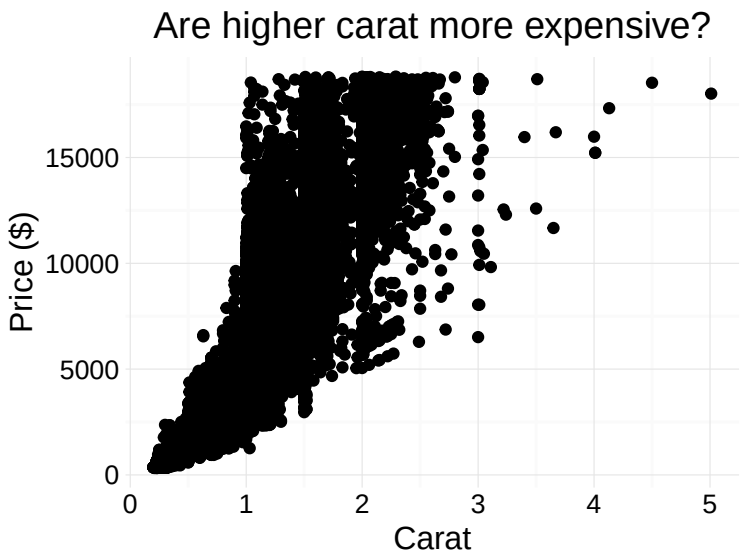
Think of the plots as being built up as layers. First we map the variables with the `ggplot` statement, then we tell ggplot what type of plot to make (*e.g.* scatter, histogram, bar plot, *etc*).

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

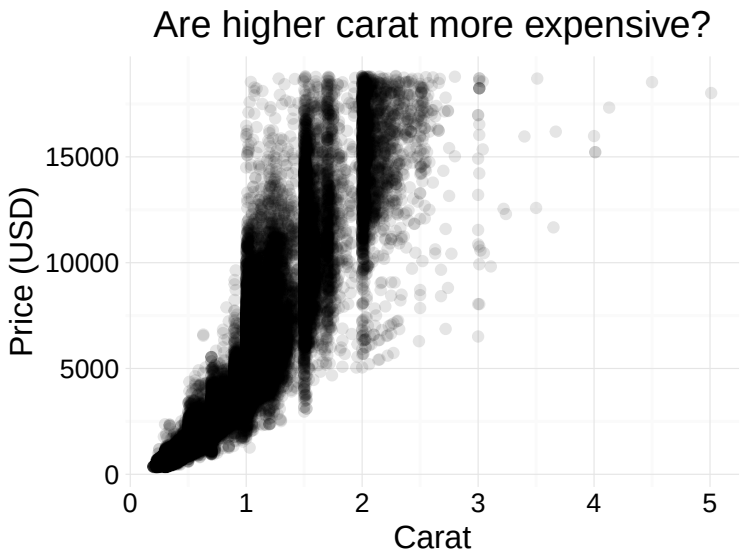


Finally, we can layer on nice labels, colors and annotation. I highly recommend having this page up when plotting: http://docs.ggplot2.org/

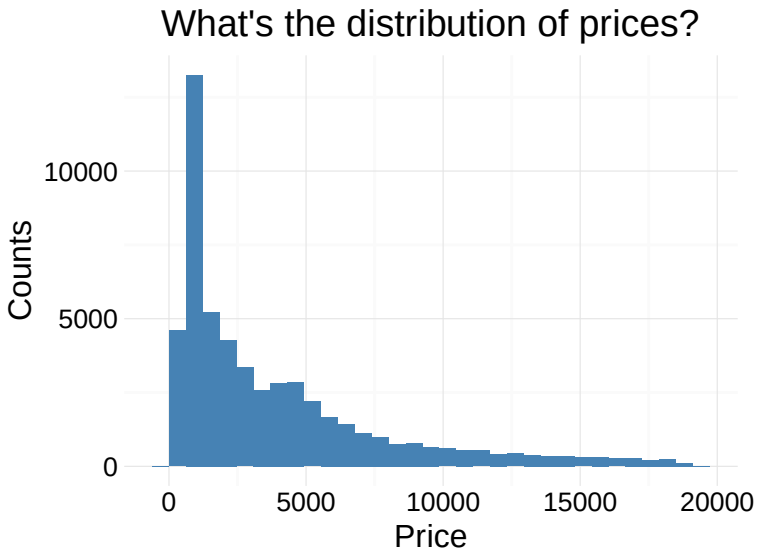
```r
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point() +
  scale_x_continuous('Carat') + # axis labels
  scale_y_continuous('Price ($)') +
  ggtitle('Are higher carat more expensive?') +
  theme_minimal() # Themes? Score! How does the plot change w/wo it?
```




```r
# what does alpha do?
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(alpha = 0.10) +
  scale_x_continuous('Carat') +
  scale_y_continuous('Price (USD)') +
  ggtitle('Are higher carat more expensive?') +
  theme_minimal()
```



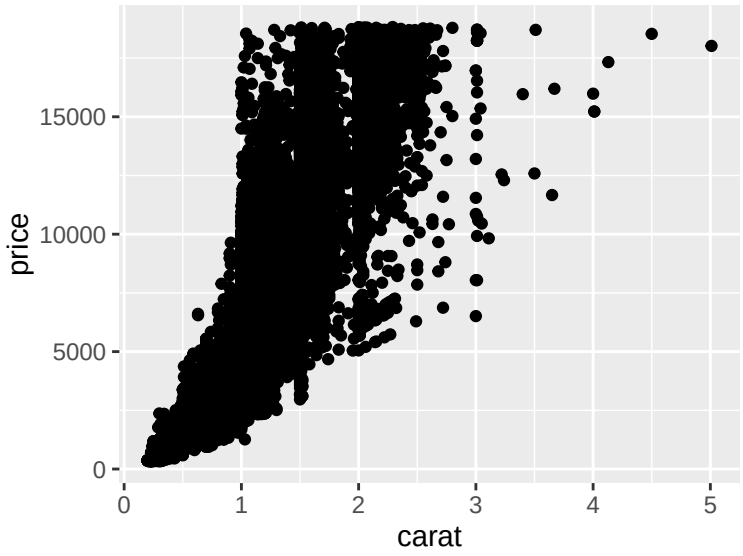
```
ggplot(diamonds, aes(x = price)) +
  geom_histogram(fill = 'steelblue') +
  scale_x_continuous('Price') +
  scale_y_continuous('Counts') +
  ggtitle("What's the distribution of prices?") +
  theme_minimal()
```



qplot

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.

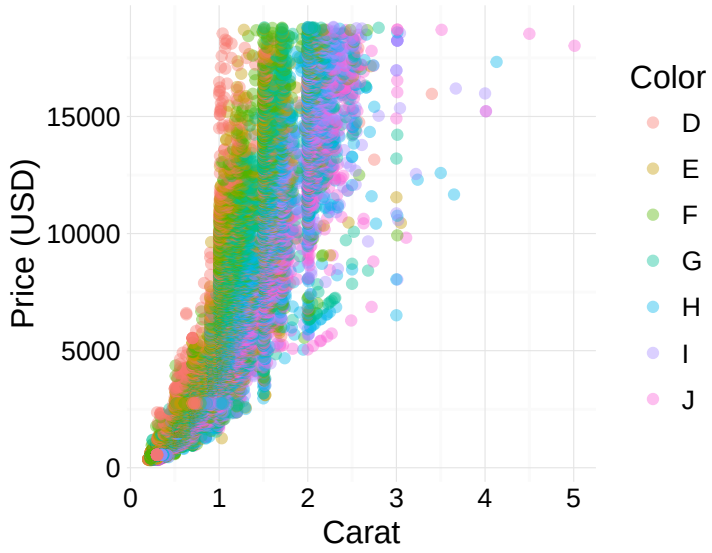
```
# quick scatterplot made with qplot
qplot(carat, price, data = diamonds)
```



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 specified the fill color explicitly in the *geom()*, when the color depends on the value of another variable, it goes inside an aesthetic, `aes()`.

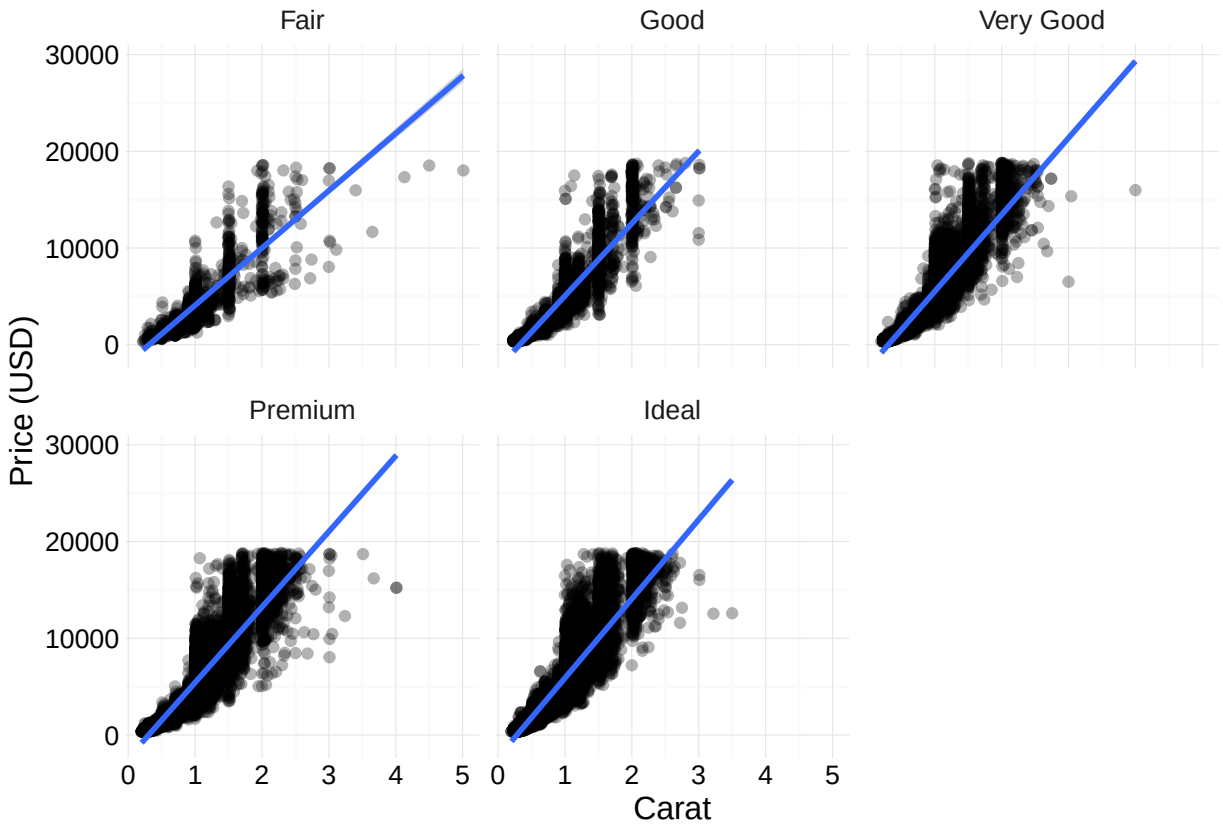
```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(aes(color = color), alpha = 0.40) +
  scale_x_continuous('Carat') +
  scale_y_continuous('Price (USD)') +
  scale_color_discrete('Color') + # legend title
  theme_minimal()
```



Faceting

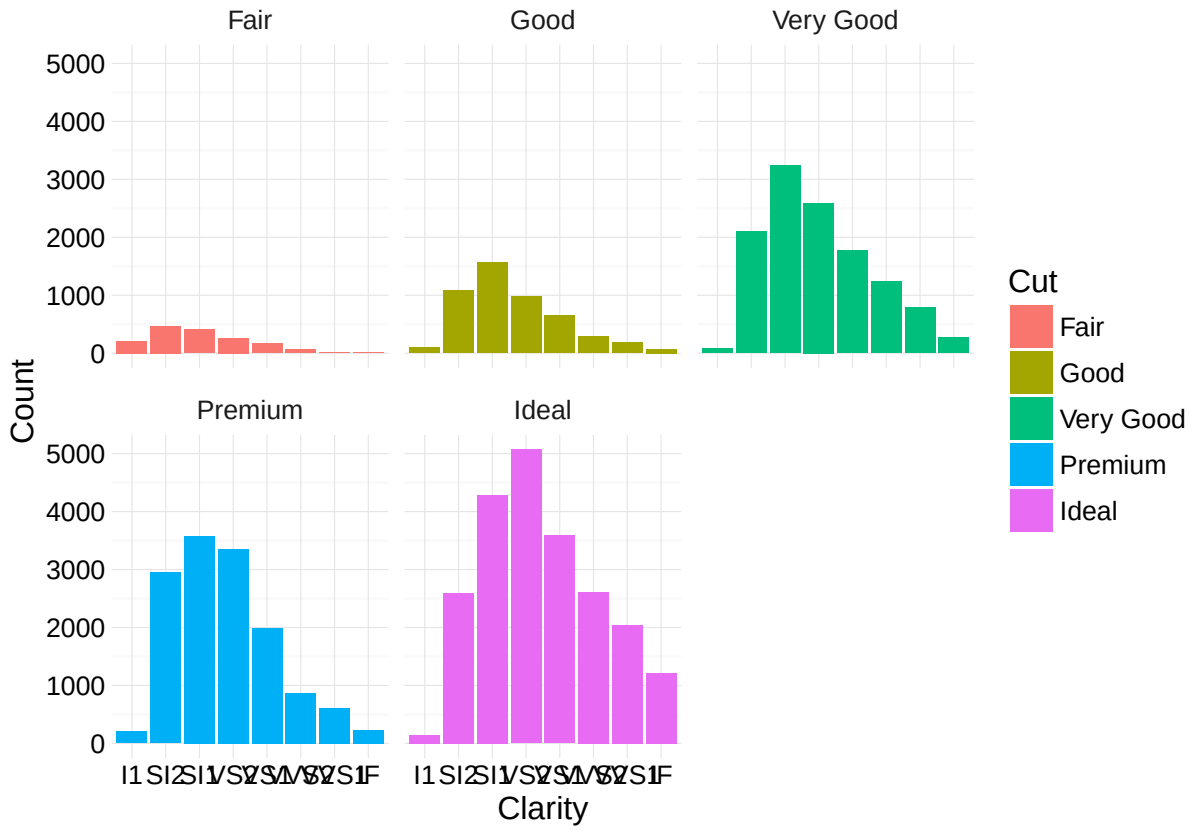
The colorful carat plot looks pretty good, but there are a lot of points laying on top of each other. When you want to compare plots across groups separately you can use a *facet* to split the figure up by category.

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(alpha = 0.30) +
  geom_smooth(method = 'lm') +
  facet_wrap(~ cut) + # split plot up by '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()
```

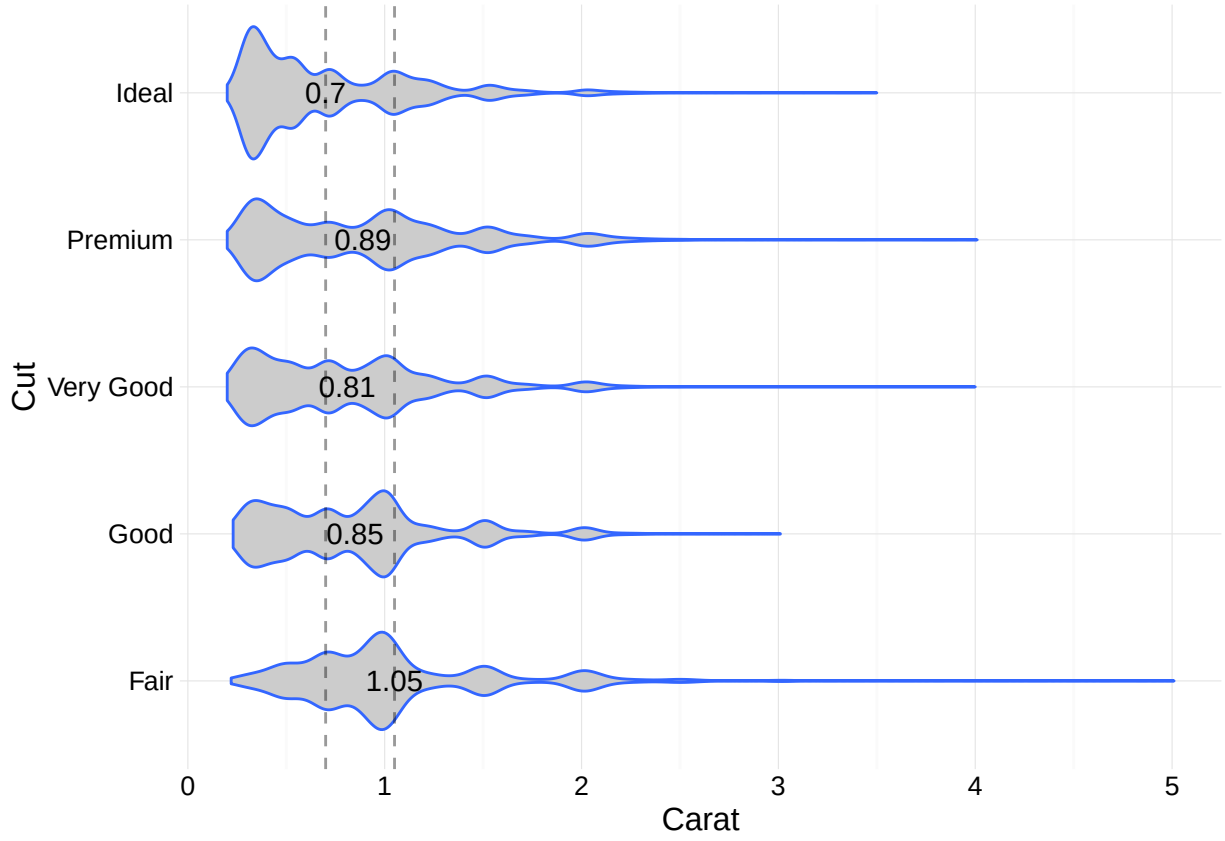


Annotation

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.

```
# summarize with dplyr (more later)
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() + # what is this doing?
  scale_x_discrete('Cut') +
  scale_y_continuous('Carat') +
  scale_fill_discrete('Cut') +
  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

Return to the `mtcars` data set we used previously. Use *ggplot2* to make visualizations that answer the following:

1. What is the relationship between weight and MPG?
2. Is that relationship modified by the number of cylinders?
3. What is the relationship between horsepower and MPG, and is that relationship modified by the number of cylinders?

Aggregation and summarization with *dplyr*

What is *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()`

Using %>% pipes in your code

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 that helps makes code more readable.

Whenever you see a `%>%` in R code, you can just read it out loud as "then".

`filter` example

The verb `filter` allows us to select one or more rows from a dataset, usually by the value of a certain variable (columns). Consider the diamonds dataset again, suppose we only wanted to look at diamonds that cost more than $10,000. We could enter the following code:

```
diamonds %>% filter(price > 10000)
```

You can read this code aloud as "Take the diamonds dataset *then* filter rows where price > $10,000." Instead of just printing this to your screen, you could also save this as a new dataset.

```
expensive_diamonds = diamonds %>% filter(price > 10000)
```

`select` example

The verb `select` allows us to select one or more columns from a dataset. Columns will generally correspond to variables in our data. You might use this if you are working with a dataset with a lot of variables and you want to limit what you are working on.

```
diamonds %>% select(cut)
```

You can read this code aloud as "Take the diamonds dataset *then* select the cut variable". We can also select multiple columns:

```
diamonds %>% select(cut, carat)
```

"Take the diamonds dataset *then* select the carat and cut variables."

Combining `select` and `filter`

Let's create a new dataset called expensive_diamonds where price > $10,000 and it only contains the following variables: cut and carat.

```
expensive_diamonds = diamonds %>% filter(price > 10000) %>% select(cut, carat)
```

"Take the diamonds dataset *then* select rows where price is greater than $10,000 *then* select the cut and carat variables." Note: you will get a different result if switch the order of `filter` and `select` in this example. Why?

Using `summarize`

`summarize` is an important verb that reduces the data, usually by aggregating multiple observations to single summary statistics.

Let's calculate the mean carat in the diamonds dataset. This is how it is traditionally done in R:

```
mean(diamonds$carat)
```

This can be done in with `summarize` with the following code:

```
diamonds %>% summarize(mean(carat))
```

"Take the diamonds dataset *then* summarize by taking the mean of carat." Compared to the traditional approach, the output looks slightly different. With `summarize` you can also name the result:

```
diamonds %>% summarize(carat_mean = mean(carat))
```

It may not seem obvious why this approach is better than the traditional approach. One reason is that `summarize` allows you to do multiple summary statistics at once, here we will calculate mean carat and mean price together (as an added bonus we can also save it as a new dataset):

```
diamonds %>%
  summarize(
    carat_mean = mean(carat),
    mean_price = mean(price)
    )

summary_diamond_data = diamonds %>%
  summarize(
    carat_mean = mean(carat),
    price_mean = mean(price)
    )
```

`group_by`

The `group_by` verb in *dplyr* let's you breakdown a dataset into a specified group of rows. We used this earlier in a *ggplot2* example:

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

Let's break it down first: we are creating a new dataset called carat_summary where we take diamonds *then* we group by the variable cuts *then* we summarize by taking the mean of carat (calling that carat_mean). By grouping by cuts, we are calculating the mean for every value of cut. What are those values?

```
unique(diamonds$cut)
```

```
## [1] Ideal     Premium   Good      Very Good Fair
## Levels: Fair < Good < Very Good < Premium < Ideal
```

So we were able to calculate the mean carat value for each type of cut. For a diamond, which type of cut has the highest average carat value?

Revisit the annotation example above: why was it useful to utilize `group_by` in that example? Note: the `round` function was applied so only two digits would be displayed on the plot.

Self-check

Return to the `mtcars` data set we used previously. Get used to coding with the *dplyr* verbs by completing the following tasks:

1. Select only the `mpg` column from `mtcar`.
2. Select rows from `mtcar` where `hp` > 110.
3. What is the the mean `mpg`?
4. The `vs` variable takes two values: 0 and 1. Calculate the mean of `mpg` when `vs` $= 0$ and when `vs` $= 1$. Hint: use `group_by` and your answer to 3.

Exploratory data analysis

Capital Bikeshare data

The *mtcars* and *diamonds* data sets were 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, season, 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?

We'll talk much more about how to do each of these steps next week and throughout the course. For now let's 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?

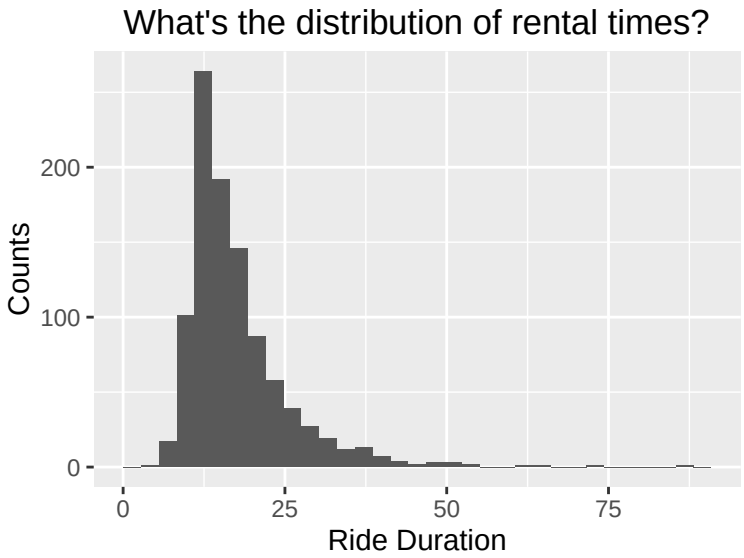
```
# How many stations are there?
length(unique(data$station))
```

```
## [1] 252
```

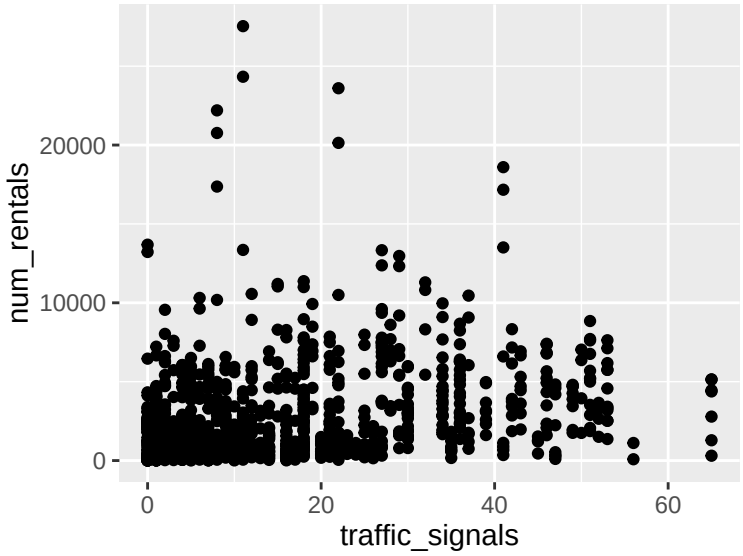
```
# What's the ride duration distribution like?
summary(data$duration_mean)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.695  12.540  15.190  17.560  20.060  87.320
```

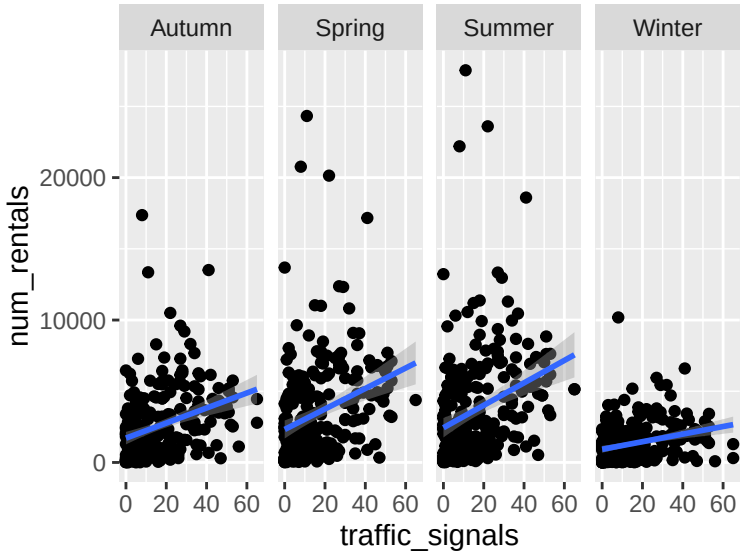
```
ggplot(data, aes(x = duration_mean)) +
  geom_histogram() + #
  scale_x_continuous('Ride Duration') +
  scale_y_continuous('Counts') +
  ggtitle("What's the distribution of rental times?")
```



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



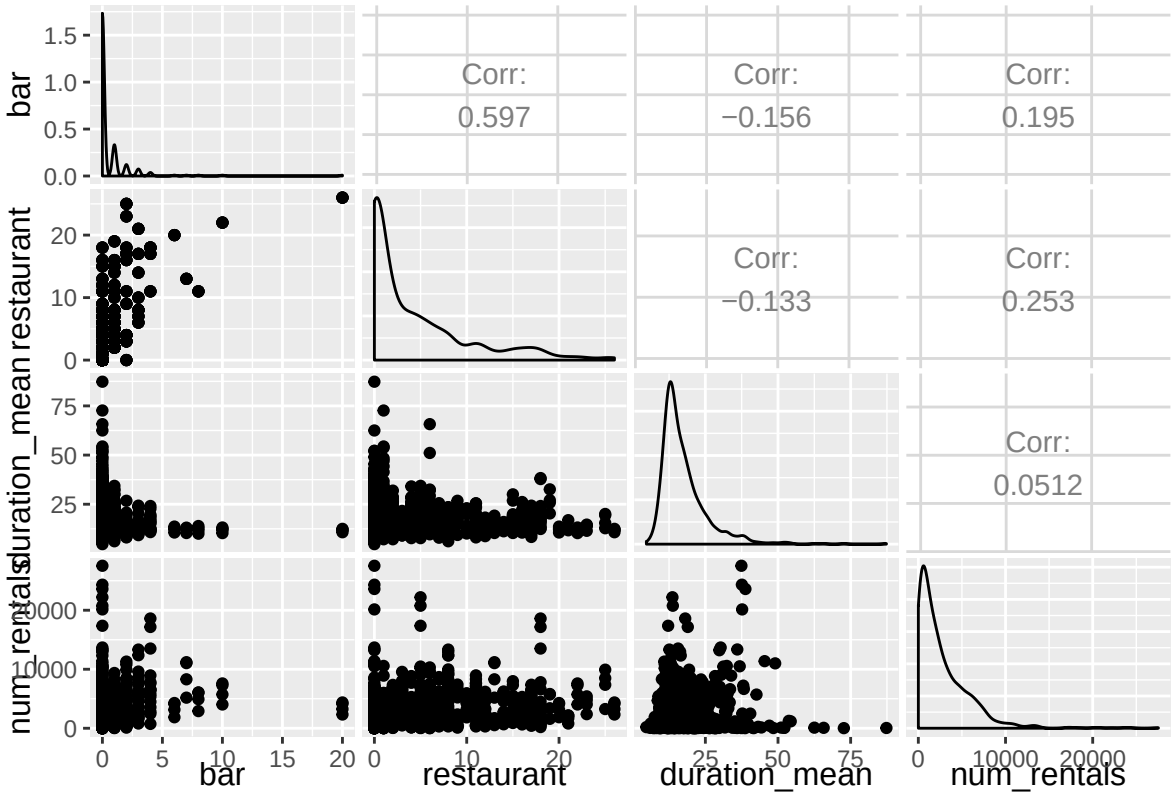

```
ggplot(data, aes(x = traffic_signals, y = num_rentals)) +
  geom_point() +
  facet_grid(. ~ season) +
  geom_smooth(method = 'lm')
```



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 1 - Exploring data with visualization

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?

** Save your code! **

Assignment 2 - Asking exploratory questions and summarizing the data

Now let's do some exploratory statistical analysis. Try to use the *dplyr* verbs.

1. Overall, what is the average (mean) `duration_mean` and `num_rentals` in the data?
2. How many rows are there where there are greater than 3000 total rentals (`num_rentals`)? What percentage is this of the total rows in the full data? (Hint: You can see how many rows and columns are in a dataset using `dim`. RStudio also shows you in the Environment tab in the upper right).
3. What is the average (mean) `duration_mean` for rows where the total number of rentals is greater than 3,000?
    - Is this number different than what you calculated in question 1?
    - Did you expect the average duration to increase, decrease or stay the same when there are more total rentals at a station?
4. Which `season` has the highest average (mean) `duration_mean`? `num_rentals`? (Hint: use `group_by`)

** Save your code! **