Lab 1

PSTAT 131/231

Aarti Garaye

Setup

R and RStudio

All work in this course, including homework, labs, and the final project, will be conducted using *R* and *RStudio*. I understand that not all students will already be familiar with *R*, so I don't expect that everyone starts out at the same coding level. The instructional team, myself included, are here to help!

First, go to https://www.r-project.org/ and click *Download R*. Select a CRAN mirror link. Do this **EVEN IF** you already have R installed on your machine. If you have a previous R installation, re-downloading R will *update* your copy of R to the most recent version, which often fixes many small problems.

Next, go to https://www.rstudio.com/products/rstudio/download/ and download the **free** version of RStudio Desktop. We will almost always open and use RStudio to interact with R.

You will be working with RStudio a lot, and you'll have time to learn most of the bells and whistles RStudio provides. Think about RStudio as your "workbench". Keep in mind that RStudio is MORE than plain R. RStudio is an environment that makes it easier to work with R, while handling many of the little tasks than can be a hassle.

At this point, your TA will give a brief overview of the RStudio default four-pane layout and demonstrate how to change fonts, settings, etc.

Getting Help with R Much of the time we spend using R involves interacting with functions. For example, to find the average of three numbers, we can call the `mean()` function:

```
mean(c(1, 2, 3))
```

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

Each function in R has its own set of arguments and possible values that these arguments accept. You will often need to look up a specific function or one of its arguments – very often! The good news is, there is a lot of R documentation out there, and it's fairly easy to get help.

To get help about `mean()`, you can uncomment (delete the `#`) and run either of these lines:

```
# ?mean
# help(mean)
```

Or simply open your Web browser and do a search for something like `R function mean help`.

The *tidyverse* and *tidymodels* Throughout this course, in the homework and labs, we'll spend a lot of time using the *tidyverse* and *tidymodels*. These are two collections of R packages that not only work together very well but also are relatively easy to use. The *tidyverse* makes a lot of data manipulation, exploring, and visualizing much simpler, and *tidymodels* has provided a framework that allows users to fit machine learning models in R more easily than ever before.

I recommend loading the following packages for all your homework assignments and for your final project. We also load a few extra packages here for use later on.

What do you think `tidymodels_prefer()` might do? Try looking it up to find out (or asking your TA)!

```
library(tidyverse)
library(tidymodels)
library(ggplot2)
library(corrplot)
library(ggthemes)
tidymodels_prefer()
```

Recall that the first time you use the packages, you'll need to install them using `install.packages()`; make sure to install any packages **outside** of an .Rmd file, though, because including that command in an .Rmd will prevent the file from knitting. Speaking of .Rmd files:

Update an .Rmd

Markdown is the language R uses to create and update documents. If you write in a Markdown file within a **code chunk**, as shown below, that text will be processed and run like R code. If you write *outside* of the code chunks, like here, that text will not be run and will appear as text. You can format it as usual, include headings, etc.!

.Rmd files are a special type of file, referred to as a dynamic document, that allows users to combine narrative (text) with R code. Because you will be turning in all your work for this course as .Rmd files and their knitted .html or .pdf file(s), it is important that you quickly become familiar with this resource.

Try updating the code in the following code chunk. Assign `2+2` to another object, called `y`. `<-` is the assignment operator in R, commonly read as "gets."

```
# This is a code chunk!
# Any uncommented text in here will be run as R code.
# For example:
x <- seq(1, 10, 1)
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Take some time and work through the Markdown tutorial here: www.markdown-tutorial.com.

In Markdown, code chunks can have specific options set for them; you can also set the options for chunks in the entire document. At the top of this .Rmd, you'll see a code chunk with `opts_chunk$set()`. Any options you set inside that function will apply to all code chunks in the document. I recommend you set the options used in this file for all your assignments, along with the options at the **very** top of the document — `toc: true`, `toc_float: true`, and `code_folding: show`. You can go further and customize Markdown files as much as you like, but that's not required.

Creating an R Project

I **strongly** recommend working in R within the context of an R project. It sounds complicated or unnecessary at first, but an R project – which is essentially a special working directory, designated with an (automatically created) `.Rproj` file – can make your life **much** easier, especially when working on your final project.

Your TA can now go over how to create a new project.

Working in an R project automatically sets your working directory to that project folder, rather than whatever your computer's default working directory is. That means you can readily access other .R scripts, photos, data files, etc. simply by putting them in your project folder, without having to write out lengthy file paths.

Basics of Data Processing

Now we'll take some time to go over some of the basic tools for managing data via the *tidyverse*. There are many more functions that you might find useful, and you can read more about them in R for Data Science, if you're interested.

First, you'll need to install and load some packages. These include, but are not limited to: `tidyverse`, `tidymodels`, and `ISLR`. Make sure to install each of these using the `install.packages()` function and load them with `library()`.

```
library(tidyverse)
library(tidymodels)
library(ISLR)
```

Some packages include datasets when they are loaded. Set `eval = TRUE` and knit your .Rmd to run the following code chunk:

```
mpg
```

Run `?mpg` to learn more about this data set.

There are five key `tidyverse` functions, or "verbs." We'll go through each of them briefly with the `mpg` data set. All of these functions work similarly; their first argument is a data frame, subsequent arguments describe operations on the data frame, and the function's result is a new data frame.

Select observations by their value: `filter()`

Say that you are interested in selecting only those rows in `mpg` that represent Audi compact cars. The easiest way to select them is:

```
mpg %>%
  filter(class == "compact" & manufacturer == "audi")
```

```
## # A tibble: 15 x 11
##    manufacturer model      displ  year   cyl trans drv     cty   hwy fl    class
##    <chr>        <chr>      <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
##  1 audi         a4           1.8  1999     4 auto~ f        18    29 p     comp~
##  2 audi         a4           1.8  1999     4 manu~ f        21    29 p     comp~
##  3 audi         a4           2    2008     4 manu~ f        20    31 p     comp~
##  4 audi         a4           2    2008     4 auto~ f        21    30 p     comp~
```

```
##  5 audi         a4           2.8  1999     6 auto~ f        16    26 p     comp~
##  6 audi         a4           2.8  1999     6 manu~ f        18    26 p     comp~
##  7 audi         a4           3.1  2008     6 auto~ f        18    27 p     comp~
##  8 audi         a4 quattro   1.8  1999     4 manu~ 4        18    26 p     comp~
##  9 audi         a4 quattro   1.8  1999     4 auto~ 4        16    25 p     comp~
## 10 audi         a4 quattro   2    2008     4 manu~ 4        20    28 p     comp~
## 11 audi         a4 quattro   2    2008     4 auto~ 4        19    27 p     comp~
## 12 audi         a4 quattro   2.8  1999     6 auto~ 4        15    25 p     comp~
## 13 audi         a4 quattro   2.8  1999     6 manu~ 4        17    25 p     comp~
## 14 audi         a4 quattro   3.1  2008     6 auto~ 4        17    25 p     comp~
## 15 audi         a4 quattro   3.1  2008     6 manu~ 4        15    25 p     comp~
```

The above code takes the `mpg` data set and pipes it into `filter()`. The pipe symbol is `%>%`; a shortcut for typing it is Cmd+Shift+M on Macs, or Cntrl+Shift+M on Windows.

If you want to store the result of your filtering, you need to assign it to an object:

```
filtered_mpg <- mpg %>%
  filter(class == "compact" & manufacturer == "audi")
```

You can use the classic comparison operators – `!=` for not equal to, `==` for equal to, `>`, etc. They can also be used in combination with Boolean operators, as demonstrated above; `&` for "and", `|` for "or", and `!` for "not."

Activities: On your own, find ways to filter the `flights` data set from the `nycflights13` package to achieve each of the following:

- Had an arrival delay of two or more hours
- Flew to Houston (IAH or HOU)
- Were operated by United, American, or Delta
- Departed in summer (July, August, and September)
- Arrived more than two hours late, but didn't leave late
- Were delayed by at least an hour, but made up over 30 minutes in flight
- Departed between midnight and 6am (inclusive)

Solution: Follow the code below. Before that, it is important to note the format of the `flights` dataset. You can follow this link but I have added it in this lab for convinience.

Format Data frame with columns

- year, month, day: Date of departure.
- dep_time, arr_time: Actual departure and arrival times (format HHMM or HMM), local tz.
- sched_dep_time, sched_arr_time: Scheduled departure and arrival times (format HHMM or HMM), local tz.
- dep_delay, arr_delay: Departure and arrival delays, in minutes. Negative times represent early departures/arrivals.
- carrier: Two letter carrier abbreviation. See airlines to get name.
- flight: Flight number.
- tailnum: Plane tail number. See planes for additional metadata.

- origin, dest: Origin and destination. See airports for additional metadata.
- air_time: Amount of time spent in the air, in minutes.
- distance: Distance between airports, in miles.
- hour, minute: Time of scheduled departure broken into hour and minutes.
- time_hour: Scheduled date and hour of the flight as a POSIXct date. Along with origin, can be used to join flights data to weather data.

```
library(nycflights13)

# Had an arrival delay of two or more hours
filtered_flights_arr_delay <- flights %>%
  filter(arr_delay >= 120) # since arrival delay is in minutes in the dataset

# Flew to Houston (IAH or HOU)
filtered_flights_to_houston <- flights %>%
  filter(dest == "IAH" | dest == "HOU")

# Operated by United, American, or Delta
filtered_flights_un_am_dt <- flights %>%
  filter(carrier == "UA" | carrier == "AA" | carrier == "DL")

# Departed in summer (July, August, and September)
filtered_flights_dept_summer <- flights %>%
  filter(month == 7 | month == 8 | month == 9)

# Arrived more than two hours late, but didn't leave late
filtered_flights_arrlate_deptnotlate <- flights %>%
  filter(arr_delay > 120 & dep_delay <= 0)

# Were delayed by at least an hour, but made up over 30 minutes in flight
filtered_flights_delayOnePlusHour_made30mins <- flights %>%
  filter(dep_delay >= 60 & (dep_delay - arr_delay) > 30)

# Departed between midnight and 6am (inclusive)
filtered_flights_midnight_6 <- flights %>%
  filter(dep_time >= 0 & dep_time <= 600) # dep_time, arr_time are in HHMM or HMM format
```

Select specific variables or columns by their names: `select()`

Often in machine learning, we end up working with very large data sets that have a lot of columns. The `mpg` data set is pretty small, but we can still practice with it.

We can select the `year`, `hwy`, and `class` variables and store them in a new object, `mpg_small`, by:

```
mpg_small <- mpg %>%
  select(year, hwy, class)
```

For a shortcut, when working with large data frames, we can use `(year:class)` or `-(year:class)` to select or de-select all columns including them and between them, respectively.

Note that we use the `head()` function here so that only a few rows of the resulting tibble are displayed when we knit to .html.

```r
mpg %>% select(year:class) %>%
  head()
```

```
## # A tibble: 6 x 8
##    year   cyl trans      drv     cty   hwy fl    class
##   <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
## 1  1999     4 auto(l5)   f        18    29 p     compact
## 2  1999     4 manual(m5) f        21    29 p     compact
## 3  2008     4 manual(m6) f        20    31 p     compact
## 4  2008     4 auto(av)   f        21    30 p     compact
## 5  1999     6 auto(l5)   f        16    26 p     compact
## 6  1999     6 manual(m5) f        18    26 p     compact
```

```r
mpg %>% select(-(year:class)) %>%
  head()
```

```
## # A tibble: 6 x 3
##   manufacturer model displ
##   <chr>        <chr> <dbl>
## 1 audi         a4      1.8
## 2 audi         a4      1.8
## 3 audi         a4      2
## 4 audi         a4      2
## 5 audi         a4      2.8
## 6 audi         a4      2.8
```

The tidyverse includes a number of helper functions that can be used inside `select()`, like `starts_with()`, etc. You can see more of them with `?select`.

Activities On your own, working with the `flights` data:

- Find as many ways as you can to select `dep_time`, `dep_delay`, `arr_time`, and `arr_delay`.
- What happens if you include the name of a variable multiple times in a `select()` call?

Create or add new variables: `mutate()`

Besides selecting existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

`mutate()` always adds new columns at the end of your dataset, so we'll use `select()` to reorder the columns and put the new ones at the front. `everything()` is a helper function to grab all the other variables.

We can add a new column that has the value `0` for cars manufactured before 2000 and `1` for those manufactured after 2000 with the following code. Variables set up in this way are "dummy-coded."

```r
mpg %>%
  mutate(after_2k = if_else(year <= 2000, 0, 1)) %>%
  select(after_2k, year, everything()) %>%
  head()
```

```
## # A tibble: 6 x 12
##   after_2k  year manufacturer model displ   cyl trans    drv     cty   hwy fl
##      <dbl> <int> <chr>        <chr> <dbl> <int> <chr>    <chr> <int> <int> <chr>
## 1        0  1999 audi         a4      1.8     4 auto(l5) f        18    29 p
## 2        0  1999 audi         a4      1.8     4 manual(~ f        21    29 p
## 3        1  2008 audi         a4      2       4 manual(~ f        20    31 p
## 4        1  2008 audi         a4      2       4 auto(av) f        21    30 p
## 5        0  1999 audi         a4      2.8     6 auto(l5) f        16    26 p
## 6        0  1999 audi         a4      2.8     6 manual(~ f        18    26 p
## # i 1 more variable: class <chr>
```

You can see an overview of a number of useful variable creation functions here: https://r4ds.had.co.nz/transform.html#mutate-funs.

For an alternative to `mutate()` when you only want to retain the newly created variables, not all variables, use `transmute()`:

```
transmute(mpg,
  after_2k = if_else(year <= 2000, 0, 1)) %>%
  head()
```

```
## # A tibble: 6 x 1
##   after_2k
##      <dbl>
## 1        0
## 2        0
## 3        1
## 4        1
## 5        0
## 6        0
```

Activities On your own, working with the `flights` data:

- Currently `dep_time` and `sched_dep_time` are convenient to look at, but hard to compute with because they're not really continuous numbers. Convert them to a more convenient representation of number of minutes since midnight.
- What does 1:3 + 1:10 return? Why do you think it returns this?

Create grouped summaries of data frames: `summarise()`

The last key verb function is `summarise()`. It's most useful when combined with `group_by()`, so that it produces a summary for each level or value of a variable/group. Notice what happens if used without grouping:

```
mpg %>%
  summarise(avg_hwy = mean(hwy))
```

```
## # A tibble: 1 x 1
##   avg_hwy
##     <dbl>
## 1    23.4
```

This value represents the average highway mileage across *all cars in the data frame*. We can see immediately that, while this has certainly reduced the size of the data frame, it's not very useful. Instead, we might prefer the average highway mileage by class of car, or by manufacturer. We can view these, and even **arrange()** by highway mileage:

```
mpg %>%
  group_by(class) %>%
  summarise(avg_hwy = mean(hwy)) %>%
  arrange(avg_hwy)
```

```
## # A tibble: 7 x 2
##   class      avg_hwy
##   <chr>        <dbl>
## 1 pickup        16.9
## 2 suv           18.1
## 3 minivan       22.4
## 4 2seater       24.8
## 5 midsize       27.3
## 6 subcompact    28.1
## 7 compact       28.3
```

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(avg_hwy = mean(hwy)) %>%
  arrange(avg_hwy)
```

```
## # A tibble: 15 x 2
##    manufacturer avg_hwy
##    <chr>          <dbl>
##  1 land rover      16.5
##  2 lincoln         17
##  3 jeep            17.6
##  4 dodge           17.9
##  5 mercury         18
##  6 ford            19.4
##  7 chevrolet       21.9
##  8 nissan          24.6
##  9 toyota          24.9
## 10 subaru          25.6
## 11 pontiac         26.4
## 12 audi            26.4
## 13 hyundai         26.9
## 14 volkswagen      29.2
## 15 honda           32.6
```

The following code finds the average highway mileage by manufacturer, counts the number of cars produced by each manufacturer, and prints the top 10 manufacturers with largest numbers of cars, arranged by mileage:

```
mpg %>%
  group_by(manufacturer) %>%
  summarise(avg_hwy = mean(hwy),
            count = n()) %>%
  filter(count >= 9) %>%
  arrange(avg_hwy)
```

```
## # A tibble: 10 x 3
##    manufacturer avg_hwy count
##    <chr>          <dbl> <int>
##  1 dodge           17.9    37
##  2 ford            19.4    25
##  3 chevrolet       21.9    19
##  4 nissan          24.6    13
##  5 toyota          24.9    34
##  6 subaru          25.6    14
##  7 audi            26.4    18
##  8 hyundai         26.9    14
##  9 volkswagen      29.2    27
## 10 honda           32.6     9
```

It's not demonstrated here, but you can also use other verbs like `mutate()` and `filter()` in conjunction with `group_by()`. Use `ungroup()` when you want to return to ungrouped data.

Exploratory Data Analysis

The last section of this lab will guide you through practicing exploratory data analysis on the dataset `diamonds` (contained in the `ggplot2` package).

Diamonds

We'll start with the `diamonds` data set. First, let's take a look at the first few lines of it, to get a feel for it:

```r
diamonds %>%
  head()
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <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
## 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

Think about which of these variables we might want to predict with a machine learning model. `price` makes intuitive sense; it's not something we can simply directly measure from a diamond, and we would likely be very interested in knowing how much a given diamond is worth.

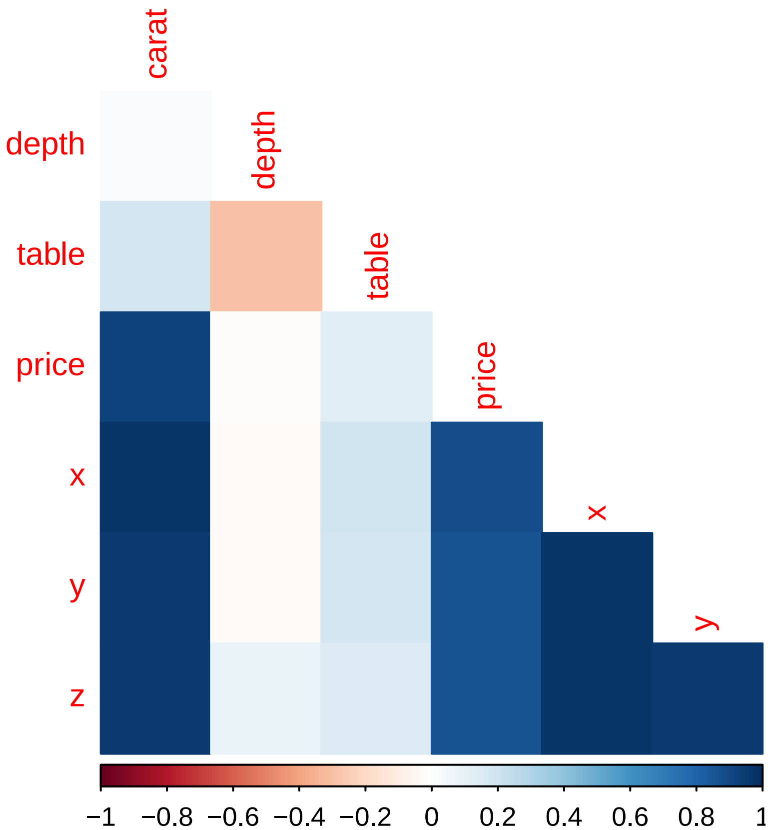
Activities:

- How many observations are there in `diamonds`?
- How many variables? Of these, how many are features we could use for predicting `price`?

Run `?diamonds` and look at the variable definitions.

First, let's make a correlation matrix to see which continuous variables are correlated with `price`. See example code below:

```
diamonds %>%
  select(is.numeric) %>%
  cor() %>%
  corrplot(type = 'lower', diag = FALSE,
           method = 'color')
```



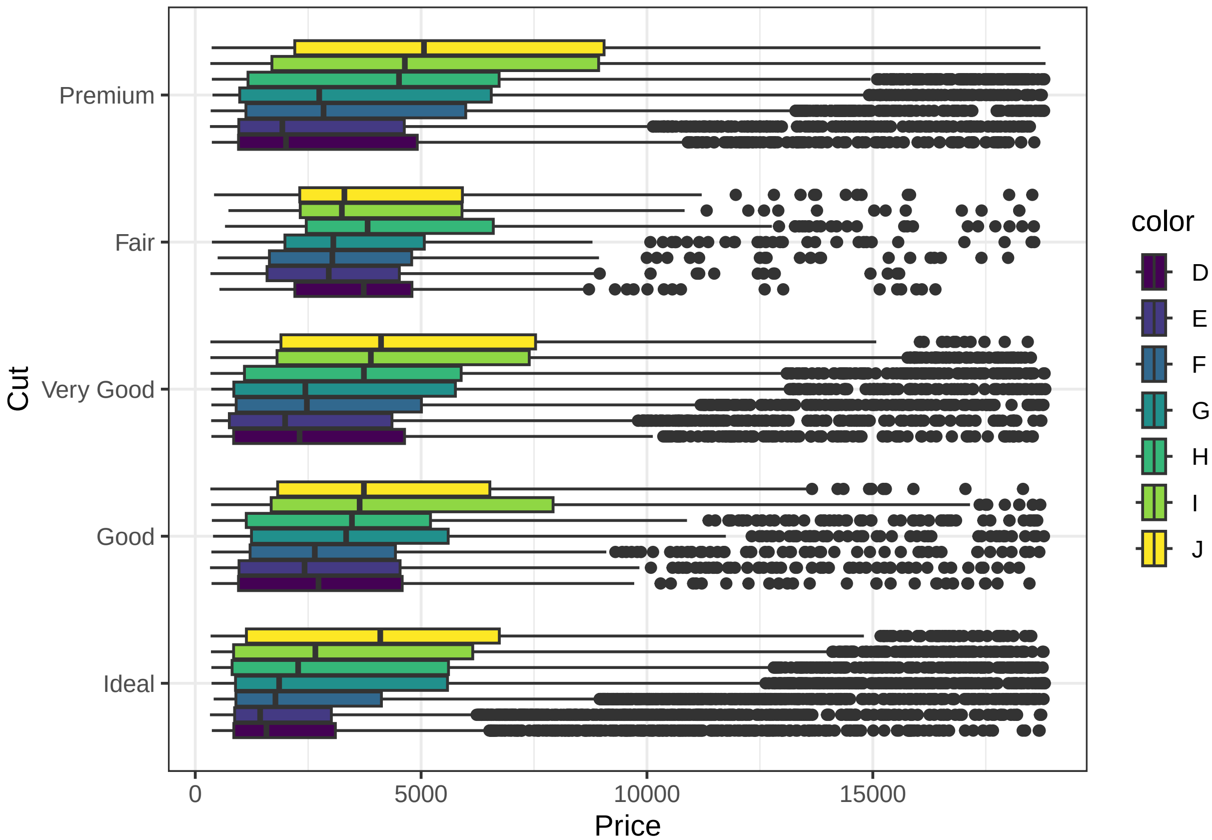
Take a moment and look at the arguments in the `corrplot()` function. What does each one do? What happens if you change `diag` to `TRUE` and `method` to `'square'`?

Activities:

- Which features are positively correlated with `price`? Do these make sense?
- Are any features negatively correlated with `price`?
- Which features are correlated with *each other*? Why do you think this might be?

Let's make a boxplot of the distribution of `price` per level of `cut` and `color`, to see if there appears to be a relationship between it and these predictors.

```
diamonds %>%
  ggplot(aes(x = price, y = reorder(cut, price), fill = color)) +
  geom_boxplot() +
  labs(y = "Cut", x = "Price") +
  theme_bw()
```



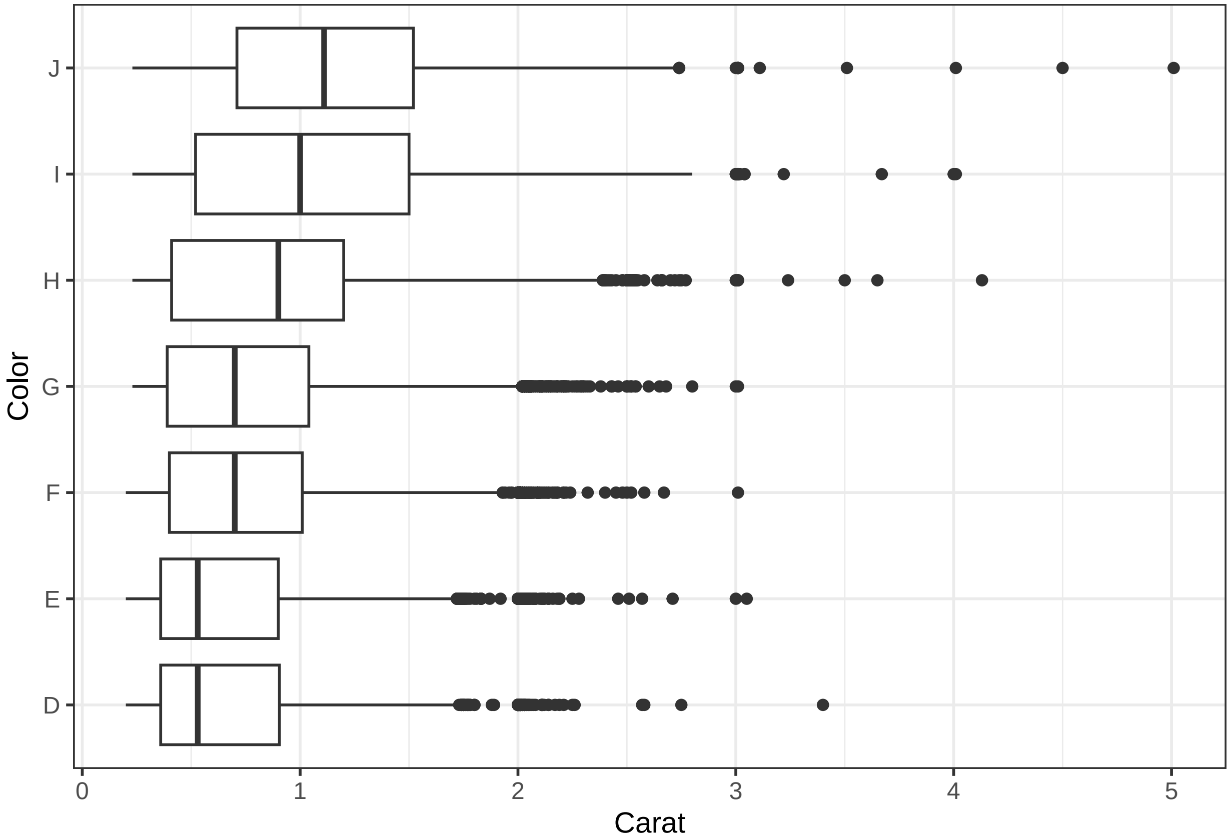
Activities:

- What do you learn from this plot about the relationship between `price`, `cut`, and `color`?
- Refer back to the definitions of the variables in `?diamonds`. Does anything you learned surprise you?

Since `J` is the worst color for diamonds, why do you think they tend to cost more?

Let's take a look at the relationship between `color` and `carat` to explore further. Remember from the correlation plot that `carat` is highly positively correlated with `price`.

```
diamonds %>%
  ggplot(aes(x = carat, y = reorder(color, carat))) +
  geom_boxplot() +
  theme_bw() +
  labs(x = "Carat", y = "Color")
```

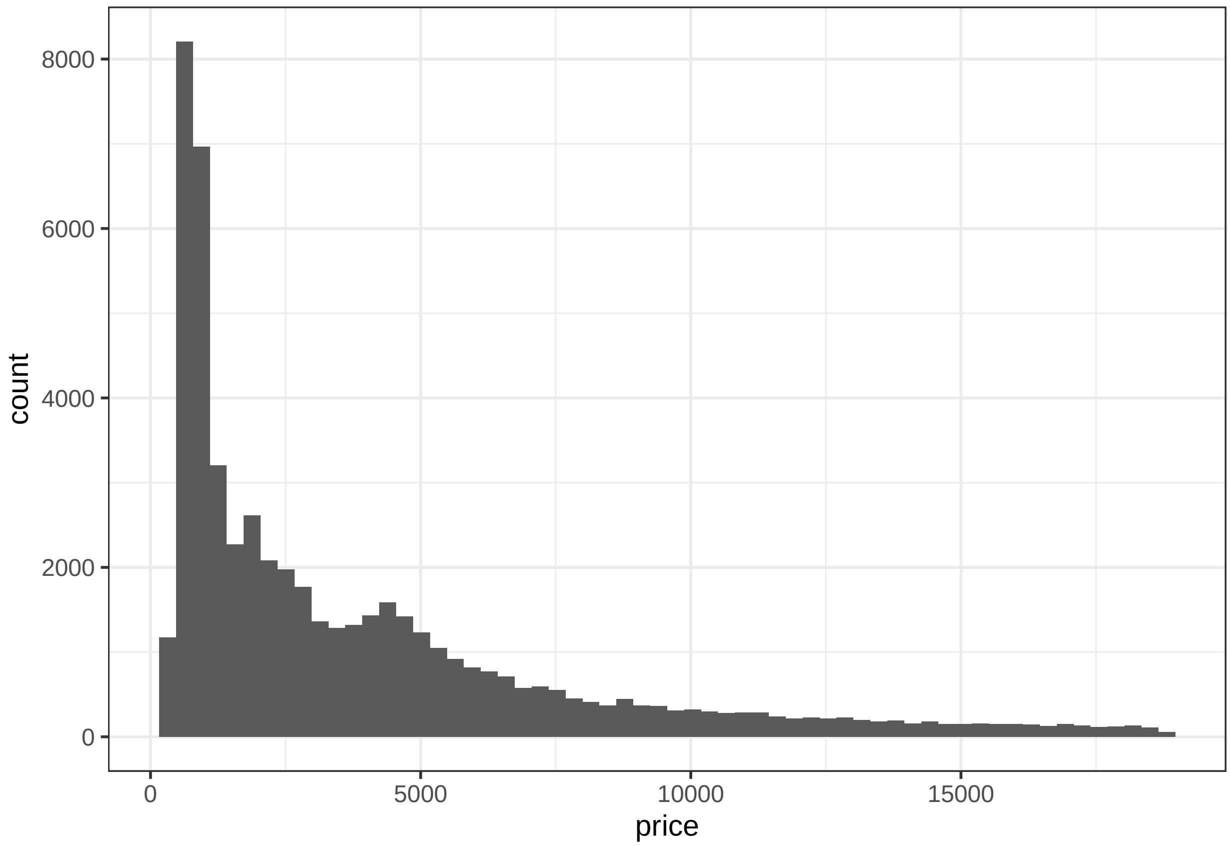


Activities:

- Explain why lower-quality colors tend to cost more.

Now we'll assess the distribution of our outcome variable `price`. Let's make a histogram:

```
diamonds %>%
  ggplot(aes(x = price)) +
  geom_histogram(bins = 60) +
  theme_bw()
```



Notice that we increased the number of bins here; this allows us to get a more fine-grained picture of the distribution. `price` is positively skewed, meaning that much of the mass of its distribution is at the lower end, with a long tail to the right. Most diamonds in the data set are worth less than $\$10,000$.

Activities:

- Create a single plot to visualize the relationship between `cut`, `carat`, and `price`.

Data Splitting

Now we're going to walk through the process of splitting the `diamonds` data set into two, a training set and a test set. Note that in the future, after we discuss the concept of resampling, we'll use a resampling technique called cross-validation, but for now, we'll work with the entire training set.

We also could have performed this split prior to doing exploratory data analysis, and in future we'll split data first. That's arguably better practice because it means we will have never encountered the test observations before we fit a final model to them.

The textbook(s) describe a way to split data using base R functions, but `tidymodels` makes the process a lot easier.

We set a seed first because the splitting process is random. If we don't set a seed, each time we re-run the code we'll get a new random split, and the results will not be identical.

In general, set a seed to whatever number you like. People often use birthdates, anniversaries, or lucky numbers, etc. Just make sure you remember the number, because you'll need to set the seed to that number to reproduce your split in future.

```r
set.seed(3435)

diamonds_split <- initial_split(diamonds, prop = 0.80,
                                strata = price)
diamonds_train <- training(diamonds_split)
diamonds_test <- testing(diamonds_split)
```

Activities:

- How many observations are now in the training and testing sets, respectively? Report the exact number, not a proportion.
- What do you think the `strata = price` argument does? Take a guess, then use `?initial_split` to verify.