An Introductory R Programming Guide

Brad Boehmke

2017-06-02

Contents

Introduction to R

A language for data analysis and graphics. This definition of R was used by Ross Ihaka and Robert Gentleman in the title of their 1996 paper[1] outlining their experience of designing and implementing the R software. It's safe to say this remains the essence of what R is; however, it's tough to encapsulate such a diverse programming language into a single phrase.

During the last decade, the R programming language has become one of the most widely used tools for statistics and data science. Its application runs the gamut from data preprocessing, cleaning, web scraping and visualization to a wide range of analytic tasks such as computational statistics, econometrics, optimization, and natural language processing. In 2012 R had over 2 million users and continues to grow by double digit percentage points every year. R has become an essential analytic software throughout industry; being used by organizations such as Google, Facebook, New York Times, Twitter, Etsy, Department of Defense, and even in presidential political campaigns.

So what makes R such a popular tool?

Open Source

R is an *open source* software created over 20 years ago by Ihaka and Gentleman at the University of Auckland, New Zealand. However, its history is even longer as its lineage goes back to the S programming language created by John Chambers out of Bell Labs back in the 1970s.[2] R is actually a combination of S with lexical scoping semantics inspired by Scheme.[3] Whereas the resulting language is very similar in appearance to S, the underlying implementation and semantics are derived from Scheme. Unbeknownst to many the S language has been a popular vehicle for research in statistical methodology, and R provides an *open source* route to participate in that activity.

Although the history of S and R is interesting[4], the principal artifact to observe is that R is an *open source* software. Although some contest that open-source software is merely a "craze" [5], most evidence suggests that open-source is here to stay and represents a *new*[6] norm for programming languages. Open-source software such as R blurs the distinction between developer and user which provides the ability to extend and modify the analytic functionality to your, or your organization's needs. The data analysis process is rarely restricted to just a handful of tasks with predictable input and outputs that can be pre-defined by a fixed user interface as is common in proprietary software. Rather, as previously mentioned in the introduction, data analysis includes unique, different, and often multiple requirements regarding the specific tasks involved. Open source software allows more flexibility for you, the data analyst, to manage how data are being transformed, manipulated, and modeled "under the hood" of software rather than relying on "stiff" point and click software interfaces. Open source also allows you to operate on every major platform rather than be restricted to what your personal budget allows or the idiosyncratic purchases of organizations.

This invariably leads to new expectations for data analysts; however, organizations are proving to greatly value the increased technical abilities of open source data analysts as evidenced by a recent O'Reilly survey revealing that data analysts focusing on open source technologies make more money than those still dealing in proprietary technologies.

---

[1] Ihaka, R., & Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, 5(3), 299-314.

[2] Consequently, R is named partly after its authors (Ross and Robert) and partly as a play on the name of S.

[3] Morandat, Frances; Hill, Brandon (2012). Evaluating the design of the R language: objects and functions for data analysis. *ECOOP'12 Proceedings of the 26th European conference on Object-Oriented Programming.*

[4] See Roger Peng's R programming for Data Science for further, yet concise, details on S and R's history.

[5] This was recently argued by Pollack et al. which was appropriately rebutted by Boehmke & Jackson. See my post which provides both articles.

[6] Open-source is far from new as its been around for decades (i.e. A-2 in the 1950s, IBM's ACP in the '60s, Tiny BASIC in the '70s) but has gained prominence since the late 1990s.

Flexibility

Another benefit of open source is that anybody can access the source code, modify and improve it. As a result, many excellent programmers contribute to improving existing R code and developing new capabilities. Researchers from all walks of life (academic institutions, industry, and focus groups such as RStudio and rOpenSci) are contributing to advancements of R's capabilities and best practices. This has resulted in some powerful tools that advance both statistical and non-statistical modeling capabilities that are taking data analysis to new levels.

Many researchers in academic institutions are using and developing R code to develop the latest techniques in statistics and machine learning. As part of their research, they often publish an R package to accompany their research articles[7]. This provides immediate access to the latest analytic techniques and implementations. And this research is not solely focused on generalized algorithms as many new capabilities are in the form of advancing analytic algorithms for tasks in specific domains. A quick assessment of the different task domains for which code is being developed illustrates the wide spectrum - econometrics, finance, chemometrics & computational physics, pharmacokinetics, social sciences, etc.

Powerful tools are also being developed to perform many tasks that greatly aid the data analysis process. This is not limited to just new ways to wrangle your data but also new ways to visualize and communicate data. R packages are now making it easier than ever to create interactive graphics and websites and produce sophisticated html and pdf reports. R packages are also integrating communication with high-performance programming languages such as C, Fortran, and C++ making data analysis more powerful, efficient, and posthaste than ever.

So although the analytic mantra *"use the right tool for the problem"* should always be in our prefrontal cortex, the advancements and flexibility of R is making it the right tool for many problems.

Community

The R community is fantastically diverse and engaged. On a daily basis, the R community generates opportunities and resources for learning about R. These cover the full spectrum of training - books, online courses[8], R user groups, workshops, conferences, etc. And with over 2 million users and developers, finding help and technical expertise is only a simple click away. Support is available through R mailing lists, Q&A websites[9], social media networks, and numerous blogs.

So now that you know how awesome R is, it's time to learn how to use it.

[7] See The Journal of Statistical Software and The R Journal

[8] Several platforms offer great online R courses such as Coursera, DataCamp, Udacity, and Udemy to name a few.

[9] Stack Overflow and CrossValidated are two great Q&A sources

R Basics

> *"Programming is like kicking yourself in the face, sooner or later your nose will bleed."* - Kyle Woodbury

A computer language is described by its *syntax* and *semantics*; where syntax is about the the grammar of the language and semantics the meaning behind the sentence. And jumping into a new programming language correlates to visiting a foreign country with only that 9th grade Spanish 101 class under your belt; there is no better way to learn than to immerse yourself in the environment! Although it'll be painful early on and your nose will surely bleed, eventually you'll learn the dialect and the quirks that come along with it.

Throughout this guide you'll learn much of the fundamental syntax and semantics of the R programming language; and hopefully with minimal face kicking involved. However, this first module serves to introduce you to many of the basics of R to get you comfortable. This includes:

- Installing R & RStudio
- Understanding the Console
- Getting help
- Working with packages
- Assignment & Evaluation
- R as a calculator
- Vectorization
- Styling guide

Installing R & RStudio

First, download and install R, a free software environment for statistical computing and graphics from CRAN, the Comprehensive R Archive Network. It is highly recommended to install a precompiled binary distribution for your operating system; follow these instructions:

1. Go to https://cran.r-project.org/
2. Click "Download R for Mac/Windows"
3. Download the appropriate file:
    - Windows users click Base, and download the installer for the latest R version
    - Mac users select the file R-3.X.X.pkg that aligns with your OS version
4. Follow the instructions of the installer.

Next, you will need to install RStudio's IDE (stands for integrated development environment), a powerful user interface for R. RStudio includes a text editor, so you do not have to install another stand-alone editor. Follow these instructions:

1. Go to RStudio for desktop https://www.rstudio.com/products/rstudio/download/
2. Select the install file for your OS
3. Follow the instructions of the installer.

There are other R IDE's available: Emacs, Microsoft R Open, Notepad++, etc; however, I have found RStudio to be my preferred route. When you are done installing RStudio click on the icon that looks like:

Figure 1: RStudio Icon

and you should get a window that looks like the following:

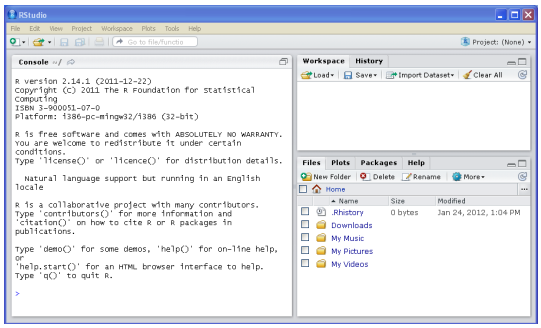


Figure 2: RStudio IDE

You are now ready to start programming!

Understanding the Console

The RStudio console is where all the action happens. There are four fundamental windows in the console, each with their own purpose. I discuss each briefly below but I highly suggest this tutorial for a thorough understanding of the console.

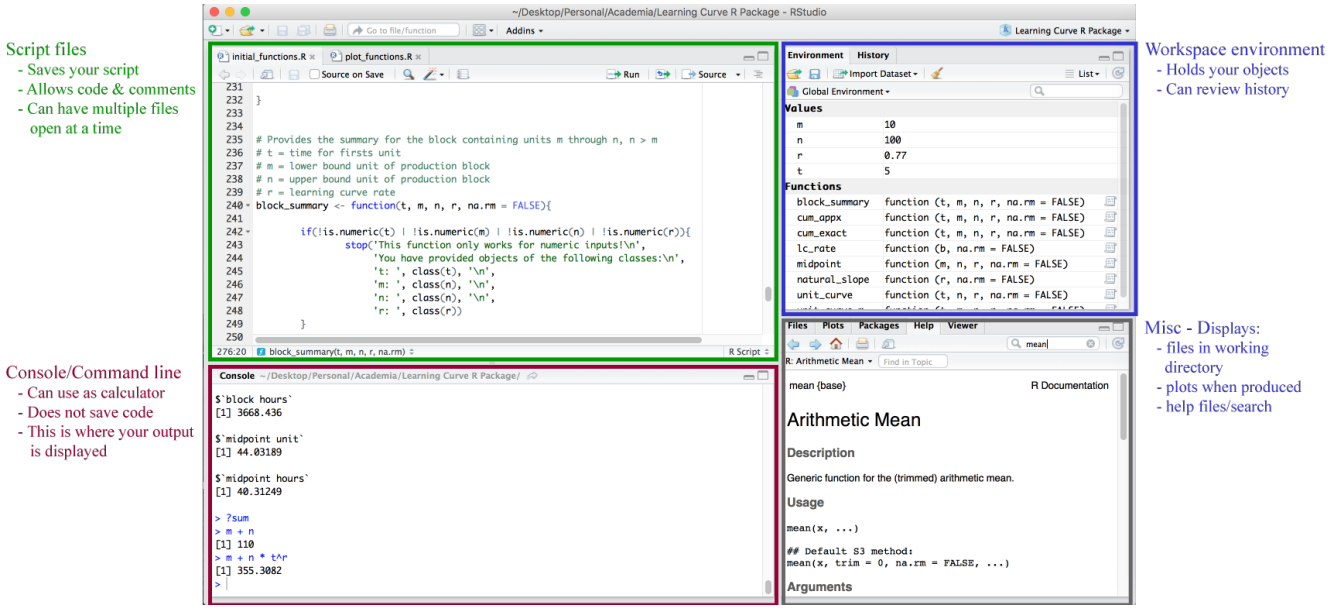


Figure 3: RStudio IDE Windows

Script Editor

The top left window is where your script files will display. There are multiple forms of script files but the basic one to start with is the .R file. To create a new file you use the **File » New File menu**. To open an existing file you use either the **File » Open File...** menu or the **Recent Files** menu to select from recently opened files. RStudio's source editor includes a variety of productivity enhancing features including syntax highlighting, code completion, multiple-file editing, and find/replace. A good introduction to the script editor can be found here.

The script editor is a great place to put code you care about. Keep experimenting in the console, but once you have written code that works and does what you want, put it in the script editor. RStudio will automatically save the contents of the editor when you quit RStudio, and will automatically load it when you re-open. Nevertheless, it's a good idea to save your scripts regularly and to back them up.

To execute the code in the script editor you have two options:

1. Place the cursor on the line that you would like to execute and execute Cmd/Ctrl + Enter. Alternatively, you could hit the **Run** button in the toolbar.
2. If you want to run *all* lines of code in the script then you can highlight the lines you want to run and then execute one of the options in #1.

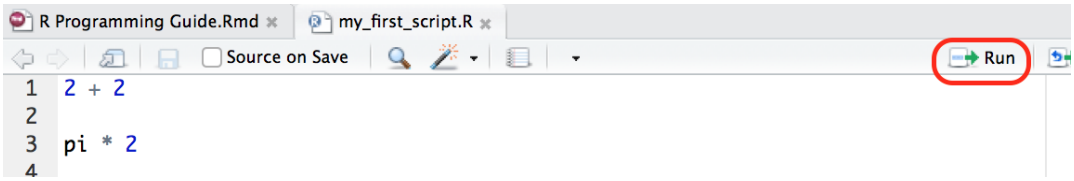


Figure 4: Running Code in the Script Editor

Workspace Environment

The top right window is the workspace environment which captures much of your current R working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions). When saving your R working session, these are the components along with the script files that will be saved in your working directory, which is the default location for all file inputs and outputs. To get or set your working directory so you can direct where your files are saved:

```
# returns path for the current working directory
getwd()

# set the working directory to a specified directory
setwd("path/of/directory")
```

For example, if I call `getwd()` the file path "/Users/bradboehmke/Desktop/Personal/Data Wrangling" is returned. If I want to set the working directory to the "Workspace" folder within the "Data Wrangling" directory I would use `setwd("Workspace")`. Now if I call `getwd()` again it returns "/Users/bradboehmke/Desktop/Personal/Data Wrangling/Workspace". An alternative solution is to go to the following location in your toolbar **Session » Set Working Directory » Choose Directory** and select the directory of choice.

The workspace environment will also list your user defined objects such as vectors, matrices, data frames, lists, and functions. For example, if you type the following in your console:

```
x <- 2
```

You will now see `x` listed in your workspace environment. To identify or remove the objects (i.e. vectors, data frames, user defined functions, etc.) in your current R environment:

```
# list all objects
ls()
## [1] "x"

# identify if an R object with a given name is present
exists("x")
## [1] TRUE
```

```r
# remove defined object from the environment
rm(x)
```

Alternatively, if you have multiple objects in your working environment you can remove more than one with:

```r
# you can remove multiple objects by using the `c()` function
rm(c("object1", "object2"))

# basically removes everything in the working environment -- use with caution!
rm(list = ls())
```

You can also view previous commands in the workspace environment by clicking the **History** tab, by simply pressing the up arrow on your keyboard, or by typing into the console:

```r
# default shows 25 most recent commands
history()

# show 100 most recent commands
history(100)

# show entire saved history
history(Inf)
```

Console

The bottom left window contains the console. You can code directly in this window but it will not save your code. It is best to use this window when you are simply wanting to perform calculator type functions. This is also where your outputs will be presented when you run code in your script. Go ahead and type the following in your console:

```r
2 * 3 + 8 / 2
## [1] 10
```

Misc. Displays

The bottom right window contains multiple tabs. The **Files** tab allows you to see which files are available in your working directory. The **Plots** tab will display any plots/graphics that are produced by your code. The **Packages** tab will list all packages downloaded to your computer and also the ones that are loaded (more on this later). And the **Help** tab allows you to search for topics you need help on and will also display any help responses (more on this later as well).

Workspace Options & Shortcuts

There are multiple options available for you to set and customize your console. You can read about, and set, available options for the current R session with the following code. For now you don't need to worry about making any adjustments, just know that *many* options do exist.

```r
# learn about available options
help(options)

# view current option settings
options()
```

```
# change a specific option (i.e. number of digits to print on output)
options(digits = 3)
```

For a thorough tutorial regarding the RStudio console and how to customize different components check out this tutorial. You can also find the RStudio console cheat sheet shown below here or by going to Help menu » Cheatsheets. As with most computer programs, there are numerous keyboard shortcuts for working with the console. To access a menu displaying all the shortcuts in RStudio you can use option + shift + k. Within RStudio you can also access them in the Help menu » Keyboard Shortcuts.

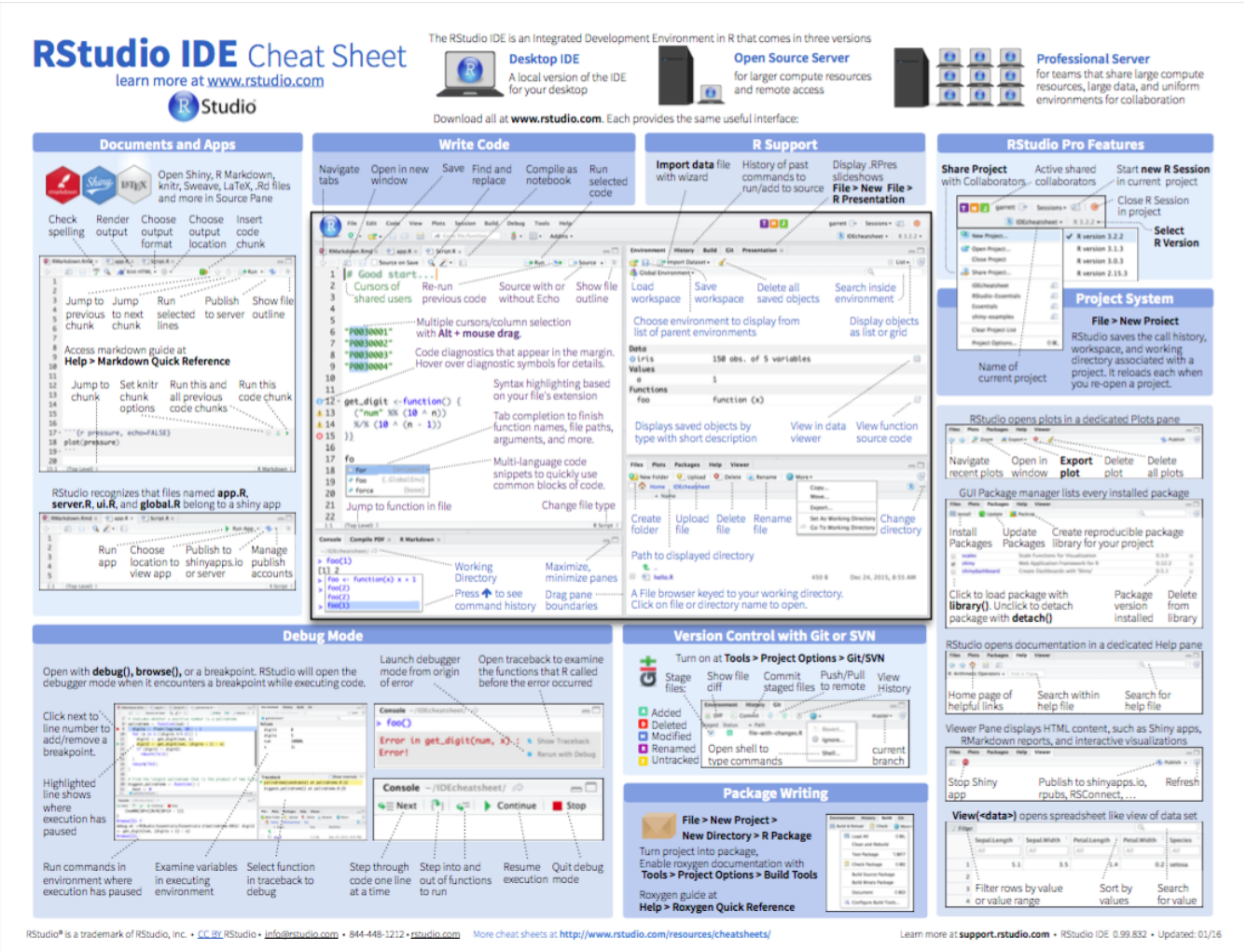


Figure 5: RStudio IDE Cheat Sheet

Exercises

1. Identify what working directory you are working out of.
2. Create a folder on your computer titled *Learning R*. Within R, set your working directory to this folder.
3. Type `pi` in the console. Set the option to show 8 digits. Re-type `pi` in the console.
4. Type `?pi` in the console. Note that documentation on this object pops up in the *Help* tab in the Misc. Display.
5. Now check out your code *History* tab.

6. Create a new .R file and save this as *my-first-script* (note how this now appears in your *Learning R* folder). Type `pi` in line 1 of this script, `option(digits = 8)` in line 2, and `pi` again in line three. Execute this code one line at a time and then re-execute all lines at once.

Getting Help

The help documentation and support in R is comprehensive and easily accessible from the command line (aka the console).

General Help

To leverage general help resources you can use:

```
# provides general help links
help.start()

# searches the help system for documentation matching a given character string
help.search("linear regression")
```

Note that the `help.search("some text here")` function requires a character string enclosed in quotation marks. So if you are in search of time series functions in R, using `help.search("time series")` will pull up a healthy list of vignettes and code demonstrations that illustrate packages and functions that work with time series data.

Getting Help on Functions

For more direct help on functions that are installed on your computer you can use the following. Test these out in your console:

```
help(mean)      # provides details for specific function
?mean           # provides same information as help(functionname)
example(mean)   # provides examples for said function
```

Note that the `help()` and `?` function calls only work for functions within loaded packages. You'll understand what this means shortly.

Getting Help from the Web

Typically, a problem you may be encountering is not new and others have faced, solved, and documented the same issue online. The following resources can be used to search for online help. Although, I typically just Google the problem and find answers relatively quickly.

- `RSiteSearch("key phrase")`: searches for the key phrase in help manuals and archived mailing lists on the R Project website.
- Stack Overflow: a searchable Q&A site oriented toward programming issues. 75% of my answers typically come from Stack Overflow.
- Cross Validated: a searchable Q&A site oriented toward statistical analysis.
- R-seek: a Google custom search that is focused on R-specific websites
- R-bloggers: a central hub of content collected from over 500 bloggers who provide news and tutorials about R.

Exercises

1. Search for documentation, code demonstrations and help pages regarding "linear regression".
2. The `stats` package is a core package that comes with base R. Pull up the help documentation regarding this package.
3. Within the `grid` package there is an introductory vignette titled "grid". Access this vignette.
4. Assume you want to make a logarithmic transformation using the `log` function. Check out the help information on this function.

Working with packages

In R, the fundamental unit of share-able code is the package. A package bundles together code, data, documentation, and tests and provides an easy method to share with others[10]. As of May 2017 there were over 10,000 packages available on CRAN, 1000 on Bioconductor, and countless more available through GitHub. This huge variety of packages is one of the reasons that R is so successful: chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package.

Installing Packages

The most common place to get packages from is CRAN. To install packages from CRAN you use `install.packages("packagename")`. For instance, if you want to install the `ggplot2` package, which is a very popular visualization package you would type the following in the console:

```
# install package from CRAN
install.packages("ggplot2")
```

As previously stated, packages are also available through Bioconductor and GitHub. Bioconductor provides R packages primarily for genomic data analyses and packages on GitHub are usually under development but have not gone through all the checks and balances to be loaded onto CRAN (aka download and use these packages at your discretion). You can learn how to install Bioconductor packages here and GitHub packages here.

Loading Packages

Once the package is downloaded onto your computer you can access the functions and resources provided by the package in two different ways:

```
# load the package to use in the current R session
library(packagename)

# use a particular function within a package without loading the package
packagename::functionname
```

For instance, if you want to have full access to the tidyr package you would use `library(tidyr)`; however, if you just wanted to use the `gather()` function which is provided by the tidyr package without fully loading tidyr you can use `tidyr::gather(...)` (here `...` just represents the arguments that you would include in this function).

[10] Wickham, H. (2015). *R packages*. "O'Reilly Media, Inc.".

Getting Help on Packages

For more direct help on packages that are installed on your computer you can use the `help` and `vignette` functions. Here we can get help on the `ggplot2` package with the following:

```
help(package = "ggplot2")       # provides details regarding contents of a package
vignette(package = "ggplot2")   # list vignettes available for a specific package
vignette("ggplot2-specs")       # view specific vignette
vignette()                      # view all vignettes on your computer
```

Note that some packages will have multiple vignettes.

Useful packages

There are thousands of helpful R packages for you to use, but navigating them all can be a challenge. To help you out, RStudio compiled a guide to some of the best packages for loading, manipulating, visualizing, analyzing, and reporting data. In addition, their list captures packages that specialize in spatial data, time series and financial data, increasing speed and performance, and developing your own R packages.

Exercises

`dplyr` is an extremely popular package for common data transformation activities and is available from CRAN. Perform the following tasks:

1. Install the `dplyr` package.
2. Load the `dplyr` package.
3. Access the help documentation for the `dplyr` package.
4. Check out the vignette(s) for `dplyr`

Assignment & Evaluation

Assignment

The first operator you'll run into is the assignment operator. The assignment operator is used to *assign* a value. For instance we can assign the value 3 to the variable `x` using the `<-` assignment operator.

```
# assignment
x <- 3
```

Interestingly, R actually allows for five assignment operators:

```
# leftward assignment
x <- value
x = value
x <<- value

# rightward assignment
value -> x
value ->> x
```

The original assignment operator in R was `<-` and has continued to be the preferred among R users. The `=` assignment operator was added in 2001 primarily because it is the accepted assignment operator in many other languages and beginners to R coming from other languages were so prone to use it. However, R uses `=` to associate function arguments with values (i.e. f(x = 3) explicitly means to call function f and set the

argument x to 3. Consequently, most R programmers prefer to keep `=` reserved for argument association and use `<-` for assignment.

The operators `<<-` is normally only used in functions which we will not get into the details. And the rightward assignment operators perform the same as their leftward counterparts, they just assign the value in an opposite direction.

Overwhelmed yet? Don't be. This is just meant to show you that there are options and you will likely come across them sooner or later. My suggestion is to stick with the tried and true `<-` operator. This is the most conventional assignment operator used and is what you will find in all the base R source code... which means it should be good enough for you.

Evaluation

We can then evaluate the variable by simply typing `x` at the command line which will return the value of `x`. Note that prior to the value returned you'll see `## [1]` in the command line. This simply implies that the output returned is the first output. Note that you can type any comments in your code by preceding the comment with the hash tag (`#`) symbol. Any values, symbols, and texts following `#` will not be evaluated.

```
# evaluation
x
## [1] 3
```

Case Sensitivity

Lastly, note that R is a case sensitive programming language. Meaning all variables, functions, and objects must be called by their exact spelling:

```
x <- 1
y <- 3
z <- 4
x * y * z
## [1] 12

x * Y * z
## Error in eval(expr, envir, enclos): object 'Y' not found
```

Exercises

1. Assign the value 5 to variable `x` (note how this shows up in your *Global Environment*).
2. Assign the character "abc" to variable `y`.
3. Evaluate the value of `x` and `y` at the command line.
4. Now use the `rm()` function to remove this object from you working environment.

R as a Calculator

Basic Arithmetic

At its most basic function R can be used as a calculator. When applying basic arithmetic, the PEMBDAS order of operations applies: **p**arentheses first followed by **e**xponentiation, **m**ultiplication and **d**ivision, and final **a**ddition and **s**ubtraction.

```
8 + 9 / 5 ^ 2
## [1] 8.36

8 + 9 / (5 ^ 2)
## [1] 8.36

8 + (9 / 5) ^ 2
## [1] 11.24

(8 + 9) / 5 ^ 2
## [1] 0.68
```

By default R will display seven digits but this can be changed using `options()` as previously outlined.

```
1 / 7
## [1] 0.1428571

options(digits = 3)
1 / 7
## [1] 0.143
```

Also, large numbers will be expressed in scientific notation which can also be adjusted using `options()`.

```
888888 * 888888
## [1] 7.9e+11

options(digits = 10)
888888 * 888888
## [1] 790121876544
```

Note that the largest number of digits that can be displayed is 22. Requesting any larger number of digits will result in an error message.

```
pi
## [1] 3.141592654

options(digits = 22)
pi
## [1] 3.141592653589793115998

options(digits = 23)
## Error in options(digits = 23): invalid 'digits' parameter, allowed 0...22
pi
## [1] 3.141592653589793115998
```

We can also perform integer divide (`%/%`) and modulo (`%%`) functions. The integer divide function will give the integer part of a fraction while the modulo will provide the remainder.

```
42 / 4          # regular division
## [1] 10.5

42 %/% 4        # integer division
## [1] 10

42 %% 4         # modulo (remainder)
## [1] 2
```

Miscellaneous Mathematical Functions

There are many built-in functions to be aware of. These include but are not limited to:

```
x <- 10

abs(10)      # absolute value
## [1] 10
sqrt(10)     # square root
## [1] 3.162278
exp(10)      # exponential transformation
## [1] 22026.47
log(10)      # logarithmic transformation
## [1] 2.302585
cos(10)      # cosine and other trigonometric functions
## [1] -0.8390715
```

Infinite, and NaN Numbers

When performing undefined calculations, R will produce `Inf` (*infinity*) and `NaN` (*not a number*) outputs.

```
1 / 0           # infinity
## [1] Inf

Inf - Inf       # infinity minus infinity
## [1] NaN

-1 / 0          # negative infinity
## [1] -Inf

0 / 0           # not a number
## [1] NaN

sqrt(-9)        # square root of -9
## Warning in sqrt(-9): NaNs produced
## [1] NaN
```

Exercises

1. Create variables `D = 1000`, `K = 5`, and `h = 0.05`.
2. Compute $2 \times D \times K$.
3. Compute $\frac{2 \times D \times K}{h}$.
4. Now put this together to compute the Economic Order Quantity, which is $\sqrt{\frac{2 \times D \times K}{h}}$. Save the output as `Q`. (hint: you'll probably need the `sqrt()` function in R)

Vectorization

Looping versus Vectorization

A key difference between R and many other languages is a topic known as vectorization. What does this mean? It means that many functions that are to be applied individually to each element in a vector of numbers require a *loop* assessment to evaluate; however, in R many of these functions have been coded in C

to perform much faster than a `for` loop would perform. For example, let's say you want to add the elements of two separate vectors of numbers (`x` and `y`).

```
x <- c(1, 3, 4)
y <- c(1, 2, 4)

x
## [1] 1 3 4
y
## [1] 1 2 4
```

In other languages you might have to run a loop to add two vectors together. In this `for` loop I print each iteration to show that the loop calculates the sum for the first elements in each vector, then performs the sum for the second elements, etc.

```
# empty vector
z <- as.vector(NULL)

# `for` loop to add corresponding elements in each vector
for (i in seq_along(x)) {
        z[i] <- x[i] + y[i]
        print(z)
}
## [1] 2
## [1] 2 5
## [1] 2 5 8
```

Instead, in R, `+` is a vectorized function which can operate on entire vectors at once. So rather than creating `for` loops for many functions, you can just use simple syntax:

```
# add each element in x and y
x + y
## [1] 2 5 8

# multiply each element in x and y
x * y
## [1]  1  6 16

# compare each element in x to y
x > y
## [1] FALSE  TRUE FALSE
```

Recycling

When performing vector operations in R, it is important to know about *recycling*. When performing an operation on two or more vectors of unequal length, R will recycle elements of the shorter vector(s) to match the longest vector. For example:

```
long <- 1:10
short <- 1:5

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

```
long + short
##  [1]  2  4  6  8 10  7  9 11 13 15
```

The elements of `long` and `short` are added together starting from the first element of both vectors. When R reaches the end of the `short` vector, it starts again at the first element of `short` and continues until it reaches the last element of the `long` vector. This functionality is very useful when you want to perform the same operation on every element of a vector. For example, say we want to multiply every element of our vector long by 3:

```
long <- 1:10
c <- 3

long * c
##  [1]  3  6  9 12 15 18 21 24 27 30
```

There are no scalars[11] in R, so `c` is actually a vector of length 1; in order to add its value to every element of `long`, it is recycled to match the length of `long`.

When the length of the longer object is a multiple of the shorter object length, the recycling occurs silently. When the longer object length is not a multiple of the shorter object length, a warning is given:

```
even_length <- 1:10
odd_length <- 1:3

even_length + odd_length
## Warning in even_length + odd_length: longer object length is not a multiple
## of shorter object length
##  [1]  2  4  6  5  7  9  8 10 12 11
```

Exercises

1. Create this vector `my_vec <- 1:10`.
2. Add 1 to every element in `my_vec`.
3. Divide every element in `my_vec` by 2.
4. Create a second vector `my_vec2 <- 10:18` and add `my_vec` to `my_vec2`.

Style Guide

> *"Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read."* - Hadley Wickham

As a medium of communication, its important to realize that the readability of code does in fact make a difference. Well styled code has many benefits to include making it easy to *i*) read, *ii*) extend, and *iii*) debug. Unfortunately, R does not come with official guidelines for code styling but such is an inconvenient truth of most open source software. However, this should not lead you to believe there is no style to be followed and over time implicit guidelines for proper code styling have been documented. What follows are guidelines that have been widely accepted as good practice in the R community and are based on Google's and Hadley Wickham's R style guides.

Notation and naming

File names should be meaningful and end with a `.R` extension.

[11] Most programming languages have *scalars* to hold a single value whereas R considers a single value to simply be a vector with one element in it.

```
# Good
weather-analysis.R
emerson-text-analysis.R

# Bad
basic-stuff.r
detail.r
```

If files need to be run in sequence, prefix them with numbers:

```
0-download.R
1-preprocessing.R
2-explore.R
3-fit-model.R
```

In R, naming conventions for variables and function are famously muddled. They include the following:

```
namingconvention        # all lower case; no separator
naming.convention       # period separator
naming_convention       # underscore separator
namingConvention        # lower camel case
NamingConvention        # upper camel case
```

Historically, there has been no clearly preferred approach with multiple naming styles sometimes used within a single package. Bottom line, your naming convention will be driven by your preference but the ultimate goal should be consistency.

My personal preference is to use all lowercase with an underscore (_) to separate words within a name. This follows Hadley Wickham's suggestions in his style guide. Furthermore, variable names should be nouns and function names should be verbs to help distinguish their purpose. Also, refrain from using existing names of functions (i.e. mean, sum, true).

Organization

Organization of your code is also important. There's nothing like trying to decipher 2,000 lines of code that has no organization. The easiest way to achieve organization is to comment your code. The general commenting scheme I use is the following.

I break up principal sections of my code that have a common purpose with:

```
#################
# Download Data #
#################
lines of code here

###################
# Preprocess Data #
###################
lines of code here

########################
# Exploratory Analysis #
########################
lines of code here
```

Then comments for specific lines of code can be done as follows:

```
code_1  # short comments can be placed to the right of code
code_2  # blah
code_3  # blah

# or comments can be placed above a line of code
code_4

# Or extremely long lines of commentary that go beyond the suggested 80
# characters per line can be broken up into multiple lines. Just don't forget
# to use the hash on each.
code_5
```

Syntax

The maximum number of characters on a single line of code should be 80 or less. If you are using RStudio you can have a margin displayed so you know when you need to break to a new line.[12] This allows your code to be printed on a normal 8.5 x 11 page with a reasonably sized font. Also, when indenting your code use two spaces rather than using tabs. The only exception is if a line break occurs inside parentheses. In this case align the wrapped line with the first character inside the parenthesis:

```
super_long_name <- seq(ymd_hm("2015-1-1 0:00"),
                       ymd_hm("2015-1-1 12:00"),
                       by = "hour")
```

Proper spacing within your code also helps with readability. The following pulls straight from Hadley Wickham's suggestions. Place spaces around all infix operators (`=`, `+`, `-`, `<-`, etc.). The same rule applies when using `=` in function calls. Always put a space after a comma, and never before.

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

There's a small exception to this rule: `:`, `::` and `:::` don't need spaces around them.

```
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

It is important to think about style when communicating any form of language. Writing code is no exception and is especially important if your code will be read by others. Following these basic style guides will get you on the right track for writing code that can be easily communicated to others.

[12] Go to *RStudio* on the menu bar then *Preferences* > *Code* > *Display* and you can select the "show margin" option and set the margin to 80.

Data File Manipulation

The first step to any data analysis process is to *get* the data. Data can come from many sources but two of the most common include text and Excel files. The first section of this module covers how to import data into R by reading data from common text files and Excel spreadsheets and shows how to view this data. For purposes of this chapter we'll use the *CustomerData_Merrimack.csv* and *CustomerData_Merrimack.xlsx* files to illustrate.

Once we have imported our data we usually want to do some initial investigation to understand the structure of our data and perform basic data manipulations such as filtering, selecting, arranging, creating, and summarizing variables. We may also want to join separate data sets and take care of missing values. The second section of this module covers these fundamental activities.

Importing Data

Reading Data from Text Files

Text files are a popular way to hold and exchange tabular data as almost any data application supports exporting data to the CSV (or other text file) formats. Text file formats use delimiters to separate the different elements in a line, and each line of data is in its own line in the text file. Therefore, importing different kinds of text files can follow a fairly consistent process once you've identified the delimiter.

Although there are base R functions to read in .csv files, the `readr` package provides importing functions that are around 10x faster than base R functions and provide more syntax consistency. To read in our .csv file we first load the `readr` package and then use `read_csv` to load the data and save it as an object titled `customer`. Within `read_csv` we simply need to provide the path to the data. In this example, I have the *CustomerData_Merrimack* data saved in a "data" sub folder in my working directory.

```
library(readr)

customer <- read_csv("data/CustomerData_Merrimack.csv")
```

`read_csv` offers many options for reading in data, which you can read about here. However, for simply importing the data set into R the default arguments work just fine. The `readr` package also offers functions to import .txt files (`read_delim()`), fixed-width files (`read_fwf()`), general text files (`read_table()`), and more.

Reading Data from Excel Files

With Excel still being the spreadsheet software of choice its important to be able to efficiently import and export data from these files. Often, R users will simply resort to exporting the Excel file as a .csv file and then import into R using `read_csv`; however, this is far from efficient. This section will teach you how to eliminate the .csv step and to import data directly from Excel.

`readxl` is one of the newest packages for accessing Excel data with R and was developed by Hadley Wickham and the RStudio team who also developed the `readr` package. This package works with both legacy .xls formats and the modern xml-based .xlsx format. Similar to `readr`, the `readxl` functions are based on a C++ library so they are extremely fast. Unlike most other packages that deal with Excel, `readxl` has no external dependencies, so you can use it to read Excel data on just about any platform.

To read in Excel data with `readxl` you will commonly use the `excel_sheets()` and `read_excel()`. `excel_sheets()` allows you to read the names of the different worksheets in the Excel workbook.

```
library(readxl)
```

```
excel_sheets("data/CustomerData_Merrimack.xlsx")
## [1] "Sheet1" "Sheet2" "Sheet3"
```

In this case, *CustomerData_Merrimack.xlsx* does not have named spreadsheets. If you look at the workbook you'll see that all the data is contained in the first spreadsheet. So we can use the `read_xlsx` function to read in this first spreadsheet.

```
customer <- read_xlsx("data/CustomerData_Merrimack.xlsx", sheet = 1)
```

Similar to `readr::read_csv`, there are many options that you can incorporate (i.e. skipping lines or columns, changing the variable names, etc.). You can check out many of these more advanced options here; however, for most basic Excel spreadsheets the default arguments will suffice.

Viewing the Data

Once you have imported the data there are several ways to get an initial view of this data prior to performing any analysis. First, you can view it in your console. This shows us that the data has 5,000 observations across 59 variables (`5000 x 59`). The first several variables that will fit in our console are printed off along with the first 10 observations. In addition, underneath the variable name you will see the *type* of data each variable is (i.e. `<chr>` for character, `<dbl>` for double).

```
customer
## # A tibble: 5,000 x 59
##         CustomerID Region TownSize Gender   Age EducationYears
##              <chr>  <dbl>    <chr>  <chr> <dbl>          <dbl>
##  1 3964-QJWTRG-NPN      1        2 Female    20             15
##  2 0648-AIPJSP-UVM      5        5   Male    22             17
##  3 5195-TLUDJE-HVO      3        4 Female    67             14
##  4 4459-VLPQUH-3OL      4        3   Male    23             16
##  5 8158-SMTQFB-CNO      2        2   Male    26             16
##  6 9662-FUSYIM-1IV      4        4   Male    64             17
##  7 7432-QKQFJJ-K72      2        5 Female    52             14
##  8 8959-RZWRHU-ST8      3        4 Female    44             16
##  9 9124-DZALHM-S6I      2        3 Female    66             12
## 10 3512-MUWBGY-52X      2        2   Male    47             11
## # ... with 4,990 more rows, and 53 more variables: JobCategory <chr>,
## #   UnionMember <chr>, EmploymentLength <dbl>, Retired <chr>,
## #   HHIncome <dbl>, DebtToIncomeRatio <dbl>, CreditDebt <dbl>,
## #   OtherDebt <dbl>, LoanDefault <chr>, MaritalStatus <chr>,
## #   HouseholdSize <dbl>, NumberPets <dbl>, NumberCats <dbl>,
## #   NumberDogs <dbl>, NumberBirds <dbl>, HomeOwner <dbl>, CarsOwned <dbl>,
## #   CarOwnership <chr>, CarBrand <chr>, CarValue <dbl>, CommuteTime <chr>,
## #   PoliticalPartyMem <chr>, Votes <chr>, CreditCard <chr>,
## #   CardTenure <dbl>, CardItemsMonthly <dbl>, CardSpendMonth <dbl>,
## #   ActiveLifestyle <chr>, PhoneCoTenure <dbl>, VoiceLastMonth <dbl>,
## #   VoiceOverTenure <chr>, EquipmentRental <chr>,
## #   EquipmentLastMonth <dbl>, EquipmentOverTenure <dbl>,
## #   CallingCard <chr>, WirelessData <chr>, DataLastMonth <dbl>,
## #   DataOverTenure <dbl>, Multiline <chr>, VM <chr>, Pager <chr>,
## #   Internet <chr>, CallerID <chr>, CallWait <chr>, CallForward <chr>,
## #   ThreeWayCalling <chr>, EBilling <chr>, TVWatchingHours <dbl>,
## #   OwnsPC <chr>, OwnsMobileDevice <chr>, OwnsGameSystem <chr>,
## #   OwnsFax <chr>, NewsSubscriber <chr>
```

Alternatively, you can use RStudio's built-in data viewer to get a scroll-able view of the complete data set

using `View` or use `str` to assess the **str**ucture of the data. Try the following:

```
View(customer)
str(customer)
```

Additional resources

In addition to text and Excel files, there are multiple other ways that data are stored and exchanged. Commercial statistical software such as SPSS, SAS, Stata, and Minitab often have the option to store data in a specific format for that software. In addition, analysts commonly use databases to store large quantities of data. R has good support to work with these additional options which we did not cover here. The following provides a list of additional resources to learn about data importing for these specific cases:

- R data import/export manual
- Working with databases
  - MySQL
  - Oracle
  - PostgreSQL
  - SQLite
  - Open Database Connectivity databases
- Importing data from commercial software
  - The `foreign` package provides functions that help you load data files from other programs such as SPSS, SAS, Stata, and others into R.

Exercises

1. Download and read in this *flights.csv* file. Can you figure out how to read in the first line to see the titles? Try read in the first 1,000 lines and only the first 6 columns (check out the help file at `?read_xlsx`.
2. Read in this *facebook.tsv* file.
3. What spreadsheets are in this *mydata.xlsx* file? Practice reading in the different spreadsheets.
4. What spreadsheets are in this *PEW Middle Class Data.xlsx* file? Can you read in the *3.Median HH income, metro* spreadsheet (hint: you'll need to skip a few lines)?

Data Frames

The *customer* data that we loaded in the previous section is being saved in R as a ***data frame***. A data frame is the most common way of storing data in R and, generally, is the data structure most often used for data analyses. Under the hood, a data frame is a list of equal-length vectors. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows. As a result, data frames can store different classes of objects in each column (i.e. numeric, character, factor). In essence, the easiest way to think of a data frame is as an Excel worksheet that contains columns of different types of data but are all of equal length rows.

Understanding the Structure

Besides viewing our data as we did in the last section we can get a quick understanding of our data frame with a few functions. First, we can get the dimensions of our data together or individually:

```
# dimensions of our data: 5000 rows x 59 columns
dim(customer)
## [1] 5000   59
```

```
# number of rows
nrow(customer)
## [1] 5000

# number of columns
ncol(customer)
## [1] 59
```

We can also check out the names of our variables with the `names` function. Go ahead and try this code in your console:

```
names(customer)
```

Creating Data Frames

Although we read in our data, which created our `customer` data frame, we can also create data frames within R with the `data.frame()` function. In this case I'll create a simple 3x4 data frame (`df`) and assess its basic structure:

```
# create the data frame
df <- data.frame(col1 = 1:3,
                 col2 = c("this", "is", "text"),
                 col3 = c(TRUE, FALSE, TRUE),
                 col4 = c(2.5, 4.2, pi))

# assess the structure of a data frame
str(df)
## 'data.frame':    3 obs. of  4 variables:
##  $ col1: int  1 2 3
##  $ col2: Factor w/ 3 levels "is","text","this": 3 1 2
##  $ col3: logi  TRUE FALSE TRUE
##  $ col4: num  2.5 4.2 3.14

# number of rows
nrow(df)
## [1] 3

# number of columns
ncol(df)
## [1] 4
```

We can also convert pre-existing structures to a data frame. The following illustrates how we can turn multiple vectors, a list, or a matrix into a data frame. Although I don't go into details in this tutorial about these different data types, you can learn about them here.

```
v1 <- 1:3
v2 <-c("this", "is", "text")
v3 <- c(TRUE, FALSE, TRUE)

# convert same length vectors to a data frame using data.frame()
data.frame(col1 = v1, col2 = v2, col3 = v3)
##   col1 col2  col3
## 1    1 this  TRUE
## 2    2   is FALSE
```

```
## 3    3 text  TRUE

# convert a list to a data frame using as.data.frame()
l <- list(item1 = 1:3, item2 = c("this", "is", "text"), item3 = c(2.5, 4.2, 5.1))
l
## $item1
## [1] 1 2 3
##
## $item2
## [1] "this" "is"   "text"
##
## $item3
## [1] 2.5 4.2 5.1

as.data.frame(l)
##   item1 item2 item3
## 1     1  this   2.5
## 2     2    is   4.2
## 3     3  text   5.1

# convert a matrix to a data frame using as.data.frame()
m1 <- matrix(1:12, nrow = 4, ncol = 3)
m1
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12

as.data.frame(m1)
##   V1 V2 V3
## 1  1  5  9
## 2  2  6 10
## 3  3  7 11
## 4  4  8 12
```

Data Transformation

Transforming your data is a basic part of data wrangling. This can include filtering, summarizing, and ordering your data by different means. This also includes combining disparate data sets, creating new variables, and many other manipulation tasks. Although many fundamental data transformation and manipulation functions exist in R, historically they have been a bit convoluted and lacked a consistent and cohesive code structure. Consequently, Hadley Wickham developed the very popular `dplyr` package to make these data processing tasks more efficient along with a syntax that is consistent and easier to remember and read.

Let's go ahead an load the `dplyr` package:

```
library(dplyr)
```

Also, we'll continue to use our `customer` data (the loaded *CustomerData_Merrimack.xlsx* data) to illustrate. In this section I'll illustrate the six key `dplyr` functions that allow you to solve the vast majority of your data manipulation challenges:

- Pick observations by their values (`filter()`).
- Pick variables by their names (`select()`).
- Reorder the rows (`arrange()`).
- Create new variables with functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarize()`).
- Combine multiple operations with the pipe operator.
- Join separate data sets (`join()`).

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire data set to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result. Let's dive in and see how these verbs work.

Filter rows with the `filter` function

`filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can filter for female customers with:

```
filter(customer, Gender == "Female")
## # A tibble: 2,494 x 59
##         CustomerID Region TownSize Gender   Age EducationYears
##              <chr>  <dbl>    <chr>  <chr> <dbl>          <dbl>
##  1 3964-QJWTRG-NPN      1        2 Female    20             15
##  2 5195-TLUDJE-HVO      3        4 Female    67             14
##  3 7432-QKQFJJ-K72      2        5 Female    52             14
##  4 8959-RZWRHU-ST8      3        4 Female    44             16
##  5 9124-DZALHM-S6I      2        3 Female    66             12
##  6 5621-QSZPSF-NF2      4        1 Female    59             19
##  7 8241-PWPONH-62O      2        4 Female    33              8
##  8 9205-PAZEXY-90Q      2        1 Female    72             20
##  9 4225-PZZDIY-IBH      3        1 Female    66             13
## 10 0758-EQEGIQ-3OF      1        1 Female    57             17
## # ... with 2,484 more rows, and 53 more variables: JobCategory <chr>,
## #   UnionMember <chr>, EmploymentLength <dbl>, Retired <chr>,
## #   HHIncome <dbl>, DebtToIncomeRatio <dbl>, CreditDebt <dbl>,
## #   OtherDebt <dbl>, LoanDefault <chr>, MaritalStatus <chr>,
## #   HouseholdSize <dbl>, NumberPets <dbl>, NumberCats <dbl>,
## #   NumberDogs <dbl>, NumberBirds <dbl>, HomeOwner <dbl>, CarsOwned <dbl>,
## #   CarOwnership <chr>, CarBrand <chr>, CarValue <dbl>, CommuteTime <chr>,
## #   PoliticalPartyMem <chr>, Votes <chr>, CreditCard <chr>,
## #   CardTenure <dbl>, CardItemsMonthly <dbl>, CardSpendMonth <dbl>,
## #   ActiveLifestyle <chr>, PhoneCoTenure <dbl>, VoiceLastMonth <dbl>,
## #   VoiceOverTenure <chr>, EquipmentRental <chr>,
## #   EquipmentLastMonth <dbl>, EquipmentOverTenure <dbl>,
## #   CallingCard <chr>, WirelessData <chr>, DataLastMonth <dbl>,
## #   DataOverTenure <dbl>, Multiline <chr>, VM <chr>, Pager <chr>,
```

```
## #   Internet <chr>, CallerID <chr>, CallWait <chr>, CallForward <chr>,
## #   ThreeWayCalling <chr>, EBilling <chr>, TVWatchingHours <dbl>,
## #   OwnsPC <chr>, OwnsMobileDevice <chr>, OwnsGameSystem <chr>,
## #   OwnsFax <chr>, NewsSubscriber <chr>
```

When you run that line of code, dplyr executes the filtering operation and returns a new data frame. dplyr functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, `<-`:

```
customer_fm <- filter(customer, Gender == "Female")
```

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).

When you're starting out with R, the easiest mistake to make is to use `=` instead of `==` when testing for equality. When this happens you'll get an informative error:

```
filter(customer, Gender = "Female")
# Error: filter() takes unnamed arguments. Do you need `==`?
```

We can add additional conditions to filter by. For example, if we want to filter for female customers that are greater than 45 in region 3:

```
filter(customer, Gender == "Female", Age > 45, Region == 3)
## # A tibble: 267 x 59
##         CustomerID Region TownSize Gender   Age EducationYears
##              <chr>  <dbl>    <chr>  <chr> <dbl>          <dbl>
##  1 5195-TLUDJE-HVO      3        4 Female    67             14
##  2 4225-PZZDIY-IBH      3        1 Female    66             13
##  3 4974-FUBHDF-Z7L      3        1 Female    77             15
##  4 5007-HXEVVL-NW5      3        2 Female    48             14
##  5 8101-YOBRYX-REF      3        3 Female    57             15
##  6 4939-KCMADL-HLM      3        2 Female    78             16
##  7 1811-FUILIP-L3S      3        1 Female    62              9
##  8 6023-TINSQC-7T4      3        3 Female    60             19
##  9 2332-AAKERL-PNS      3        4 Female    61             17
## 10 7164-HJCLUK-29N      3        2 Female    69              8
## # ... with 257 more rows, and 53 more variables: JobCategory <chr>,
## #   UnionMember <chr>, EmploymentLength <dbl>, Retired <chr>,
## #   HHIncome <dbl>, DebtToIncomeRatio <dbl>, CreditDebt <dbl>,
## #   OtherDebt <dbl>, LoanDefault <chr>, MaritalStatus <chr>,
## #   HouseholdSize <dbl>, NumberPets <dbl>, NumberCats <dbl>,
## #   NumberDogs <dbl>, NumberBirds <dbl>, HomeOwner <dbl>, CarsOwned <dbl>,
## #   CarOwnership <chr>, CarBrand <chr>, CarValue <dbl>, CommuteTime <chr>,
## #   PoliticalPartyMem <chr>, Votes <chr>, CreditCard <chr>,
## #   CardTenure <dbl>, CardItemsMonthly <dbl>, CardSpendMonth <dbl>,
## #   ActiveLifestyle <chr>, PhoneCoTenure <dbl>, VoiceLastMonth <dbl>,
## #   VoiceOverTenure <chr>, EquipmentRental <chr>,
## #   EquipmentLastMonth <dbl>, EquipmentOverTenure <dbl>,
## #   CallingCard <chr>, WirelessData <chr>, DataLastMonth <dbl>,
## #   DataOverTenure <dbl>, Multiline <chr>, VM <chr>, Pager <chr>,
## #   Internet <chr>, CallerID <chr>, CallWait <chr>, CallForward <chr>,
## #   ThreeWayCalling <chr>, EBilling <chr>, TVWatchingHours <dbl>,
## #   OwnsPC <chr>, OwnsMobileDevice <chr>, OwnsGameSystem <chr>,
## #   OwnsFax <chr>, NewsSubscriber <chr>
```

The above code finds only those observations that meet every condition; however, what if we want to filter

for one or the other. For example, what if we want to find all female customers that are either greater than 45 ***or*** live in region 3. Here we use `|` to signal "or":

```
filter(customer, Gender == "Female", Age > 45 | Region == 3)
## # A tibble: 1,503 x 59
##         CustomerID Region TownSize Gender   Age EducationYears
##              <chr>  <dbl>    <chr>  <chr> <dbl>          <dbl>
##  1 5195-TLUDJE-HVO      3        4 Female    67             14
##  2 7432-QKQFJJ-K72      2        5 Female    52             14
##  3 8959-RZWRHU-ST8      3        4 Female    44             16
##  4 9124-DZALHM-S6I      2        3 Female    66             12
##  5 5621-QSZPSF-NF2      4        1 Female    59             19
##  6 9205-PAZEXY-90Q      2        1 Female    72             20
##  7 4225-PZZDIY-IBH      3        1 Female    66             13
##  8 0758-EQEGIQ-3OF      1        1 Female    57             17
##  9 3853-NVDCOJ-TIN      1        1 Female    78             16
## 10 2041-PNMGHX-TXJ      4        3 Female    73             14
## # ... with 1,493 more rows, and 53 more variables: JobCategory <chr>,
## #   UnionMember <chr>, EmploymentLength <dbl>, Retired <chr>,
## #   HHIncome <dbl>, DebtToIncomeRatio <dbl>, CreditDebt <dbl>,
## #   OtherDebt <dbl>, LoanDefault <chr>, MaritalStatus <chr>,
## #   HouseholdSize <dbl>, NumberPets <dbl>, NumberCats <dbl>,
## #   NumberDogs <dbl>, NumberBirds <dbl>, HomeOwner <dbl>, CarsOwned <dbl>,
## #   CarOwnership <chr>, CarBrand <chr>, CarValue <dbl>, CommuteTime <chr>,
## #   PoliticalPartyMem <chr>, Votes <chr>, CreditCard <chr>,
## #   CardTenure <dbl>, CardItemsMonthly <dbl>, CardSpendMonth <dbl>,
## #   ActiveLifestyle <chr>, PhoneCoTenure <dbl>, VoiceLastMonth <dbl>,
## #   VoiceOverTenure <chr>, EquipmentRental <chr>,
## #   EquipmentLastMonth <dbl>, EquipmentOverTenure <dbl>,
## #   CallingCard <chr>, WirelessData <chr>, DataLastMonth <dbl>,
## #   DataOverTenure <dbl>, Multiline <chr>, VM <chr>, Pager <chr>,
## #   Internet <chr>, CallerID <chr>, CallWait <chr>, CallForward <chr>,
## #   ThreeWayCalling <chr>, EBilling <chr>, TVWatchingHours <dbl>,
## #   OwnsPC <chr>, OwnsMobileDevice <chr>, OwnsGameSystem <chr>,
## #   OwnsFax <chr>, NewsSubscriber <chr>
```

What if want to find all female customers that live in region 3 ***or*** region 5. There are two ways to approach this and you can see using the `%in%` operator reduces the amount of code to type. Go ahead and try these in your console.

```
# more typing
filter(customer, Gender == "Female", Region == 3 | Region == 5)

# less typing
filter(customer, Gender == "Female", Region %in% c(3,5))
```

To filter for missing values we us `is.na`. So if we want to filter for or not for those customer observations that do not have gender recorded we can use the following. Go ahead and try these in your console.

```
# filter for NAs
filter(customer, is.na(Gender))

# filter out NAs
filter(customer, !is.na(Gender))
```

Select variables with the `select` function

It's not uncommon to get data sets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

```
# select columns by name
select(customer, CustomerID, Region, TownSize, Gender)
## # A tibble: 5,000 x 4
##         CustomerID Region TownSize Gender
##              <chr>  <dbl>    <chr>  <chr>
##  1 3964-QJWTRG-NPN      1        2 Female
##  2 0648-AIPJSP-UVM      5        5   Male
##  3 5195-TLUDJE-HVO      3        4 Female
##  4 4459-VLPQUH-3OL      4        3   Male
##  5 8158-SMTQFB-CNO      2        2   Male
##  6 9662-FUSYIM-1IV      4        4   Male
##  7 7432-QKQFJJ-K72      2        5 Female
##  8 8959-RZWRHU-ST8      3        4 Female
##  9 9124-DZALHM-S6I      2        3 Female
## 10 3512-MUWBGY-52X      2        2   Male
## # ... with 4,990 more rows

# select all columns between CustomerID and Gender
select(customer, CustomerID:Gender)
## # A tibble: 5,000 x 4
##         CustomerID Region TownSize Gender
##              <chr>  <dbl>    <chr>  <chr>
##  1 3964-QJWTRG-NPN      1        2 Female
##  2 0648-AIPJSP-UVM      5        5   Male
##  3 5195-TLUDJE-HVO      3        4 Female
##  4 4459-VLPQUH-3OL      4        3   Male
##  5 8158-SMTQFB-CNO      2        2   Male
##  6 9662-FUSYIM-1IV      4        4   Male
##  7 7432-QKQFJJ-K72      2        5 Female
##  8 8959-RZWRHU-ST8      3        4 Female
##  9 9124-DZALHM-S6I      2        3 Female
## 10 3512-MUWBGY-52X      2        2   Male
## # ... with 4,990 more rows

# select all columns except those between CustomerID and Gender

select(customer, -(CustomerID:Gender))
## # A tibble: 5,000 x 55
##      Age EducationYears  JobCategory UnionMember EmploymentLength Retired
##    <dbl>          <dbl>        <chr>       <chr>            <dbl>   <chr>
##  1    20             15 Professional         Yes                0      No
##  2    22             17        Sales          No                0      No
##  3    67             14        Sales          No               16      No
##  4    23             16        Sales          No                0      No
##  5    26             16        Sales          No                1      No
##  6    64             17      Service          No               22      No
##  7    52             14 Professional          No               10      No
##  8    44             16 Professional          No               11      No
##  9    66             12 Professional          No               15     Yes
```

```
## 10    47             11        Labor          No               19      No
## # ... with 4,990 more rows, and 49 more variables: HHIncome <dbl>,
## #   DebtToIncomeRatio <dbl>, CreditDebt <dbl>, OtherDebt <dbl>,
## #   LoanDefault <chr>, MaritalStatus <chr>, HouseholdSize <dbl>,
## #   NumberPets <dbl>, NumberCats <dbl>, NumberDogs <dbl>,
## #   NumberBirds <dbl>, HomeOwner <dbl>, CarsOwned <dbl>,
## #   CarOwnership <chr>, CarBrand <chr>, CarValue <dbl>, CommuteTime <chr>,
## #   PoliticalPartyMem <chr>, Votes <chr>, CreditCard <chr>,
## #   CardTenure <dbl>, CardItemsMonthly <dbl>, CardSpendMonth <dbl>,
## #   ActiveLifestyle <chr>, PhoneCoTenure <dbl>, VoiceLastMonth <dbl>,
## #   VoiceOverTenure <chr>, EquipmentRental <chr>,
## #   EquipmentLastMonth <dbl>, EquipmentOverTenure <dbl>,
## #   CallingCard <chr>, WirelessData <chr>, DataLastMonth <dbl>,
## #   DataOverTenure <dbl>, Multiline <chr>, VM <chr>, Pager <chr>,
## #   Internet <chr>, CallerID <chr>, CallWait <chr>, CallForward <chr>,
## #   ThreeWayCalling <chr>, EBilling <chr>, TVWatchingHours <dbl>,
## #   OwnsPC <chr>, OwnsMobileDevice <chr>, OwnsGameSystem <chr>,
## #   OwnsFax <chr>, NewsSubscriber <chr>
```

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")`: matches names that begin with "abc".
- `ends_with("xyz")`: matches names that end with "xyz".
- `contains("ijk")`: matches names that contain "ijk".
- `matches("(.)\\1")`: selects variables that match a regular expression. This one matches any variables that contain repeated characters.
- `num_range("x", 1:3)` matches x1, x2 and x3.

See `?select` for more details.

`select()` can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned. Instead, use `rename()`, which is a variant of `select()` that keeps all the variables that aren't explicitly mentioned. Here we rename *CustomerID* to just *ID*:

```
rename(customer, ID = CustomerID)
## # A tibble: 5,000 x 59
##                 ID Region TownSize Gender   Age EducationYears
##              <chr>  <dbl>    <chr>  <chr> <dbl>          <dbl>
##  1 3964-QJWTRG-NPN      1        2 Female    20             15
##  2 0648-AIPJSP-UVM      5        5   Male    22             17
##  3 5195-TLUDJE-HVO      3        4 Female    67             14
##  4 4459-VLPQUH-3OL      4        3   Male    23             16
##  5 8158-SMTQFB-CNO      2        2   Male    26             16
##  6 9662-FUSYIM-1IV      4        4   Male    64             17
##  7 7432-QKQFJJ-K72      2        5 Female    52             14
##  8 8959-RZWRHU-ST8      3        4 Female    44             16
##  9 9124-DZALHM-S6I      2        3 Female    66             12
## 10 3512-MUWBGY-52X      2        2   Male    47             11
## # ... with 4,990 more rows, and 53 more variables: JobCategory <chr>,
## #   UnionMember <chr>, EmploymentLength <dbl>, Retired <chr>,
## #   HHIncome <dbl>, DebtToIncomeRatio <dbl>, CreditDebt <dbl>,
## #   OtherDebt <dbl>, LoanDefault <chr>, MaritalStatus <chr>,
## #   HouseholdSize <dbl>, NumberPets <dbl>, NumberCats <dbl>,
## #   NumberDogs <dbl>, NumberBirds <dbl>, HomeOwner <dbl>, CarsOwned <dbl>,
## #   CarOwnership <chr>, CarBrand <chr>, CarValue <dbl>, CommuteTime <chr>,
## #   PoliticalPartyMem <chr>, Votes <chr>, CreditCard <chr>,
```

```
## #   CardTenure <dbl>, CardItemsMonthly <dbl>, CardSpendMonth <dbl>,
## #   ActiveLifestyle <chr>, PhoneCoTenure <dbl>, VoiceLastMonth <dbl>,
## #   VoiceOverTenure <chr>, EquipmentRental <chr>,
## #   EquipmentLastMonth <dbl>, EquipmentOverTenure <dbl>,
## #   CallingCard <chr>, WirelessData <chr>, DataLastMonth <dbl>,
## #   DataOverTenure <dbl>, Multiline <chr>, VM <chr>, Pager <chr>,
## #   Internet <chr>, CallerID <chr>, CallWait <chr>, CallForward <chr>,
## #   ThreeWayCalling <chr>, EBilling <chr>, TVWatchingHours <dbl>,
## #   OwnsPC <chr>, OwnsMobileDevice <chr>, OwnsGameSystem <chr>,
## #   OwnsFax <chr>, NewsSubscriber <chr>
```

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame.

```
select(customer, CustomerID, CardTenure, CardSpendMonth, everything())
## # A tibble: 5,000 x 59
##         CustomerID CardTenure CardSpendMonth Region TownSize Gender   Age
##              <chr>      <dbl>          <dbl>  <dbl>    <chr>  <chr> <dbl>
##  1 3964-QJWTRG-NPN          2          816.6      1        2 Female    20
##  2 0648-AIPJSP-UVM          4          426.0      5        5   Male    22
##  3 5195-TLUDJE-HVO         35         1842.2      3        4 Female    67
##  4 4459-VLPQUH-3OL          5         3409.9      4        3   Male    23
##  5 8158-SMTQFB-CNO          8         2551.0      2        2   Male    26
##  6 9662-FUSYIM-1IV         18         2282.7      4        4   Male    64
##  7 7432-QKQFJJ-K72          3         8223.2      2        5 Female    52
##  8 8959-RZWRHU-ST8         25         5927.0      3        4 Female    44
##  9 9124-DZALHM-S6I         26         3265.9      2        3 Female    66
## 10 3512-MUWBGY-52X          2         1996.4      2        2   Male    47
## # ... with 4,990 more rows, and 52 more variables: EducationYears <dbl>,
## #   JobCategory <chr>, UnionMember <chr>, EmploymentLength <dbl>,
## #   Retired <chr>, HHIncome <dbl>, DebtToIncomeRatio <dbl>,
## #   CreditDebt <dbl>, OtherDebt <dbl>, LoanDefault <chr>,
## #   MaritalStatus <chr>, HouseholdSize <dbl>, NumberPets <dbl>,
## #   NumberCats <dbl>, NumberDogs <dbl>, NumberBirds <dbl>,
## #   HomeOwner <dbl>, CarsOwned <dbl>, CarOwnership <chr>, CarBrand <chr>,
## #   CarValue <dbl>, CommuteTime <chr>, PoliticalPartyMem <chr>,
## #   Votes <chr>, CreditCard <chr>, CardItemsMonthly <dbl>,
## #   ActiveLifestyle <chr>, PhoneCoTenure <dbl>, VoiceLastMonth <dbl>,
## #   VoiceOverTenure <chr>, EquipmentRental <chr>,
## #   EquipmentLastMonth <dbl>, EquipmentOverTenure <dbl>,
## #   CallingCard <chr>, WirelessData <chr>, DataLastMonth <dbl>,
## #   DataOverTenure <dbl>, Multiline <chr>, VM <chr>, Pager <chr>,
## #   Internet <chr>, CallerID <chr>, CallWait <chr>, CallForward <chr>,
## #   ThreeWayCalling <chr>, EBilling <chr>, TVWatchingHours <dbl>,
## #   OwnsPC <chr>, OwnsMobileDevice <chr>, OwnsGameSystem <chr>,
## #   OwnsFax <chr>, NewsSubscriber <chr>
```

Let's go ahead and reduce the number of variables that we are using for the rest of this chapter.

```
sub_cust <- select(customer, CustomerID, Region, Gender, Age,
                   HHIncome, CardSpendMonth)
```

Arrange rows with the arrange function

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes

a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(sub_cust, Age, CardSpendMonth)
## # A tibble: 5,000 x 6
##         CustomerID Region Gender   Age HHIncome CardSpendMonth
##              <chr>  <dbl>  <chr> <dbl>    <dbl>          <dbl>
##  1 3273-ZWQAKA-NK6      3   Male    18    16000          186.2
##  2 2641-SAADER-H8I      4 Female    18    16000          490.5
##  3 6158-MAYKTG-TE4      1   Male    18    13000          525.0
##  4 9723-VUGZBJ-ZQA      3   Male    18    15000          563.2
##  5 1567-HTQZHJ-ZZH      4 Female    18    28000          610.5
##  6 2895-TGIPGU-5AX      4   Male    18    23000          667.6
##  7 0224-HMKMEG-EJF      1 Female    18    13000          681.6
##  8 6152-KPRGTZ-ZVJ      5   Male    18    17000          931.3
##  9 8193-BKLGVR-4BH      1 Female    18    15000          944.4
## 10 5402-YTVDEJ-BTW      4   Male    18    15000          970.9
## # ... with 4,990 more rows
```

Use `desc()` to re-order by a column in descending order:

```
arrange(sub_cust, desc(Age), CardSpendMonth)
## # A tibble: 5,000 x 6
##         CustomerID Region Gender   Age HHIncome CardSpendMonth
##              <chr>  <dbl>  <chr> <dbl>    <dbl>          <dbl>
##  1 2521-BKEFER-MS1      3   Male    79     9000          211.0
##  2 3292-NHKPUC-JVO      5   Male    79    11000          528.3
##  3 1317-OEYPNM-WZ4      3 Female    79     9000          650.6
##  4 2175-VGCTPF-370      4   Male    79     9000          653.6
##  5 5929-MQKELC-0J1      5 Female    79     9000          742.6
##  6 5904-QLUPMU-GY6      2 Female    79    16000          788.7
##  7 4316-RMIBNS-8I3      1   Male    79    12000          813.3
##  8 4699-UUYJVC-WQ2      1   Male    79     9000          827.1
##  9 7896-NRGKAL-CX3      3 Female    79     9000          968.0
## 10 3685-WTAVBE-RC3      2 Female    79    11000          990.1
## # ... with 4,990 more rows
```

Add new variables with the `mutate` function

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`. Let's create a variable that represents a ratio of *CardSpendMonth* to *HHIncome*:

```
mutate(sub_cust, ratio = CardSpendMonth / HHIncome)
## # A tibble: 5,000 x 7
##         CustomerID Region Gender   Age HHIncome CardSpendMonth      ratio
##              <chr>  <dbl>  <chr> <dbl>    <dbl>          <dbl>      <dbl>
##  1 3964-QJWTRG-NPN      1 Female    20    31000          816.6 0.02634194
##  2 0648-AIPJSP-UVM      5   Male    22    15000          426.0 0.02840000
##  3 5195-TLUDJE-HVO      3 Female    67    35000         1842.2 0.05263429
##  4 4459-VLPQUH-3OL      4   Male    23    20000         3409.9 0.17049500
##  5 8158-SMTQFB-CNO      2   Male    26    23000         2551.0 0.11091304
##  6 9662-FUSYIM-1IV      4   Male    64   107000         2282.7 0.02133364
##  7 7432-QKQFJJ-K72      2 Female    52    77000         8223.2 0.10679481
##  8 8959-RZWRHU-ST8      3 Female    44    97000         5927.0 0.06110309
```

```
##  9 9124-DZALHM-S6I      2 Female    66    16000         3265.9 0.20411875
## 10 3512-MUWBGY-52X      2   Male    47    84000         1996.4 0.02376667
## # ... with 4,990 more rows
```

You can also create multiple new variables at the same time:

```
mutate(sub_cust,
       ratio1 = round(CardSpendMonth / HHIncome, 2),
       ratio2 = round(CardSpendMonth / Age, 2))
## # A tibble: 5,000 x 8
##         CustomerID Region Gender   Age HHIncome CardSpendMonth ratio1
##              <chr>  <dbl>  <chr> <dbl>    <dbl>          <dbl>  <dbl>
##  1 3964-QJWTRG-NPN      1 Female    20    31000          816.6   0.03
##  2 0648-AIPJSP-UVM      5   Male    22    15000          426.0   0.03
##  3 5195-TLUDJE-HVO      3 Female    67    35000         1842.2   0.05
##  4 4459-VLPQUH-3OL      4   Male    23    20000         3409.9   0.17
##  5 8158-SMTQFB-CNO      2   Male    26    23000         2551.0   0.11
##  6 9662-FUSYIM-1IV      4   Male    64   107000         2282.7   0.02
##  7 7432-QKQFJJ-K72      2 Female    52    77000         8223.2   0.11
##  8 8959-RZWRHU-ST8      3 Female    44    97000         5927.0   0.06
##  9 9124-DZALHM-S6I      2 Female    66    16000         3265.9   0.20
## 10 3512-MUWBGY-52X      2   Male    47    84000         1996.4   0.02
## # ... with 4,990 more rows, and 1 more variables: ratio2 <dbl>
```

If you only want to keep the new variables, use `transmute()`:

```
transmute(sub_cust,
       ratio1 = round(CardSpendMonth / HHIncome, 2),
       ratio2 = round(CardSpendMonth / Age, 2))
## # A tibble: 5,000 x 2
##    ratio1 ratio2
##     <dbl>  <dbl>
##  1   0.03  40.83
##  2   0.03  19.36
##  3   0.05  27.50
##  4   0.17 148.26
##  5   0.11  98.12
##  6   0.02  35.67
##  7   0.11 158.14
##  8   0.06 134.70
##  9   0.20  49.48
## 10   0.02  42.48
## # ... with 4,990 more rows
```

There are many functions for creating new variables that you can use with mutate(). The key property is that the function must be vectorized: it must take a vector of values as input, return a vector with the same number of values as output. There's no way to list every possible function that you might use, but figure 6 shows a selection of functions that are frequently useful.

Grouped summaries with the `summarize` function

Obviously the goal of all this data *wrangling* is to be able to perform statistical analysis on our data. The `summarize()` function allows us to perform the majority of summary statistics when performing exploratory data analysis.

Lets get the mean monthly card expenditure value across all customers:

lead()	ntile()	cumsum()
lag()	between()	cummax()
dense_rank()	cume_dist()	cummin()
min_rank()	cumall()	cumprod()
percent_rank()	cumany()	pmax()
row_number()	cumean()	pmin()

Figure 6: Built-in Functions for mutate()

```
summarize(sub_cust, Avg_spend = mean(CardSpendMonth, na.rm = TRUE))
## # A tibble: 1 x 1
##   Avg_spend
##       <dbl>
## 1  3372.025
```

You may have wondered about the `na.rm` argument we used above. What happens if we don't set it? You can get an `NA` in response! Although `CardSpendMonth` does not have any missing values I'll illustrate by computing the average household size of our customers:

```
# reduce our data size for easy viewing
cust_size <- select(customer, CustomerID:Age, HouseholdSize)

# compute average household size without removing missing values
summarize(cust_size, Avg_size = mean(HouseholdSize))
## # A tibble: 1 x 1
##   Avg_size
##      <dbl>
## 1       NA

# compute average household size with removing missing values
summarize(cust_size, Avg_size = mean(HouseholdSize, na.rm = TRUE))
## # A tibble: 1 x 1
##   Avg_size
##      <dbl>
## 1 2.202324
```

This is because aggregation functions obey the usual rule of missing values: if there's any missing value in the input, the output will be a missing value. Fortunately, all aggregation functions have an `na.rm` argument which removes the missing values prior to computation. Its just best to get in the practice of include `na.rm = TRUE` when computing summary statistics.

`summarize` is not terribly useful unless we pair it with `group_by()`. This changes the unit of analysis from the complete data set to individual groups. Then, when you use the dplyr verbs on a grouped data frame they'll be automatically applied "by group". For example, let's compute the mean monthly card expenditure ***grouped by*** gender:

```
by_gender <- group_by(sub_cust, Gender)
summarize(by_gender, Avg_spend = mean(CardSpendMonth, na.rm = TRUE))
## # A tibble: 3 x 2
##   Gender Avg_spend
##    <chr>     <dbl>
## 1 Female  3224.516
```

```
## 2   Male  3533.410
## 3   <NA>  2426.079
```

We see that the average monthly card expenditure for males is $3,533.41, for females is $3,224.52, and for the customers where a gender value is missing it is $2,426.08. We can even compute multiple summary statistics at once:

```
by_gender <- group_by(sub_cust, Gender)
summarize(by_gender,
          Avg_spend = mean(CardSpendMonth, na.rm = TRUE),
          Std_dev_spend = sd(CardSpendMonth, na.rm = TRUE))
## # A tibble: 3 x 3
##   Gender Avg_spend Std_dev_spend
##    <chr>     <dbl>         <dbl>
## 1 Female  3224.516      2317.035
## 2   Male  3533.410      2580.062
## 3   <NA>  2426.079      1303.684
```

We can add onto the `group_by()` statement to perform summaries at multiple levels. For example, we can compute average monthly card expenditure by gender and region:

```
by_gdr_rgn <- group_by(sub_cust, Gender, Region)
summarize(by_gdr_rgn, Avg_spend = mean(CardSpendMonth, na.rm = TRUE))
## Source: local data frame [15 x 3]
## Groups: Gender [?]
##
## # A tibble: 15 x 3
##    Gender Region Avg_spend
##     <chr>  <dbl>     <dbl>
##  1 Female      1  3022.695
##  2 Female      2  3252.470
##  3 Female      3  3201.711
##  4 Female      4  3269.212
##  5 Female      5  3383.079
##  6   Male      1  3375.981
##  7   Male      2  3439.352
##  8   Male      3  3692.818
##  9   Male      4  3535.671
## 10   Male      5  3617.054
## 11   <NA>      1  2759.333
## 12   <NA>      2  2752.517
## 13   <NA>      3  2499.350
## 14   <NA>      4  2420.575
## 15   <NA>      5  1870.750
```

Together `group_by()` and `summarise()` provide one of the tools that you'll use most commonly when working with dplyr: grouped summaries. But before we go any further with this, we need to introduce a powerful new idea: the pipe.

Combining multiple operations with the pipe function

Imagine that we want to explore the relationship between the average *CardSpendMonth* to *HHIncome* ratio for each age group for males and find the age group with the largest spend-to-income ratio. Using what you know about dplyr, you might write code like this:

```
males <- filter(sub_cust, Gender == "Male")
males <- mutate(males, ratio = CardSpendMonth / HHIncome)
males_age <- group_by(males, Age)
avg_data <- summarise(males_age, Avg_ratio = mean(ratio, na.rm = TRUE))
arrange(avg_data, desc(Avg_ratio))
## # A tibble: 62 x 2
##      Age Avg_ratio
##    <dbl>     <dbl>
##  1    20 0.1470240
##  2    18 0.1452089
##  3    79 0.1440063
##  4    19 0.1425964
##  5    24 0.1363957
##  6    75 0.1296193
##  7    78 0.1275020
##  8    21 0.1255091
##  9    28 0.1247479
## 10    74 0.1236770
## # ... with 52 more rows
```

There are five steps to prepare this data:

1. Filter for males
2. Create a new ratio variable
3. Group by age
4. Summarize to compute the average ratio
5. Arrange the data in descending order

This code is a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things is hard, so this slows down our analysis.

There's another way to tackle the same problem with the pipe, `%>%`:

```
avg_data <- sub_cust %>%
  filter(Gender == "Male") %>%
  mutate(ratio = CardSpendMonth / HHIncome) %>%
  group_by(Age) %>%
  summarise(Avg_ratio = mean(ratio, na.rm = TRUE)) %>%
  arrange(desc(Avg_ratio))

avg_data
## # A tibble: 62 x 2
##      Age Avg_ratio
##    <dbl>     <dbl>
##  1    20 0.1470240
##  2    18 0.1452089
##  3    79 0.1440063
##  4    19 0.1425964
##  5    24 0.1363957
##  6    75 0.1296193
##  7    78 0.1275020
##  8    21 0.1255091
##  9    28 0.1247479
## 10    74 0.1236770
## # ... with 52 more rows
```

This approach focuses on the transformations, not what's being transformed, which makes the code easier to read. You can read it as a series of imperative statements: take our `sub_cust` data *then* filter, *then* mutate, *then* group by, *then* summarize, *then* arrange. As suggested by this reading, a good way to pronounce `%>%` when reading code is "then".

Behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)` and so on. You can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom. We'll use piping frequently from now on because it considerably improves the readability of code. You can learn more about the pipe operator here.

Join separate data sets with the `join` functions

Often we have separate data frames that can have common and differing variables for similar observations and we wish to *join* these data frames together. `dplyr` offers multiple joining functions (`xxx_join()`) that provide alternative ways to join data frames:

- inner_join()
- left_join()
- right_join()
- full_join()

To help you learn how joins work, we'll use the following data:

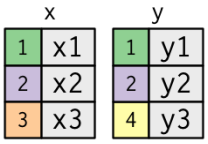


Figure 7: Data for Joins

```
x <- data_frame(
  key = 1:3,
  val_x = c("x1", "x2", "x3")
)

y  <- data_frame(
  key = c(1, 2, 4),
  val_y = c("y1", "y2", "y3")
)
```

An inner join matches pairs of observations whenever their keys are equal:

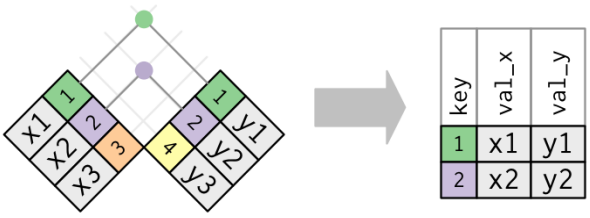


Figure 8: Inner Join

```
x %>% inner_join(y, by = "key")
## # A tibble: 2 x 3
##     key val_x val_y
```

```
##   <dbl> <chr> <chr>
## 1     1    x1    y1
## 2     2    x2    y2
```

An inner join keeps observations that appear in both tables. An outer join keeps observations that appear in at least one of the tables. There are three types of outer joins:

- A left join keeps all observations in `x`.
- A right join keeps all observations in `y`.
- A full join keeps all observations in `x` and `y`.

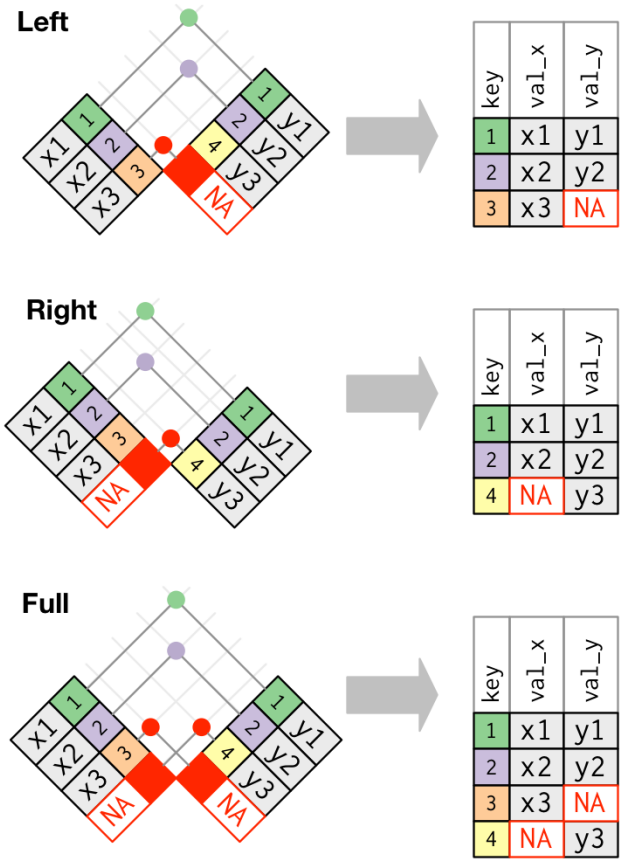


Figure 9: Outer Join Methods

Try each of the following lines of code to see how the results differ:

```
# left join
x %>% left_join(y, by = "key")

# right join
x %>% right_join(y, by = "key")

# full join
x %>% full_join(y, by = "key")
```

There are additional ways to join data to include filtering joins and set operations. You can learn more about these joining methods and managing relational data sets here.

Managing Missing Values

A common task in data analysis is dealing with missing values. In R, missing values are often represented by `NA` or some other value that represents missing values (i.e. `99`). In the previous section we saw how to filter individual variables for missing values and how to exclude them from summary statistics. In this section I illustrate how to test for, recode, and exclude missing values across your entire data frame.

Testing for missing values

To identify missing values use `is.na()` which returns a logical vector with `TRUE` in the element locations that contain missing values represented by `NA`. `is.na()` will work on vectors, lists, matrices, and data frames.

```
# identify missing values in our customer data
is.na(customer)
```

If you run the above code you'll realize that it is not very useful. However, we can quickly identify the number of missing values across our data set by wrapping `is.na` with `sum()`:

```
# how many missing values are in our data
sum(is.na(customer))
## [1] 124
```

If we want to find out where these missing values are located in our data we can run the following code, which will return only those columns with missing values and the number of missing values that exist. Here we see that our missing values are concentrated in 8 of our 59 variables.

```
missing <- colSums(is.na(customer))
sort(missing[missing > 0], decreasing = TRUE)
##   NumberBirds        Gender   JobCategory     HomeOwner HouseholdSize
##            34            33            15            13             8
##    NumberDogs    NumberCats    NumberPets
##             8             7             6
```

Recoding missing values

To re-code missing values; or re-code specific indicators that represent missing values, we can use normal subsetting and assignment operations. For example, you may not have noticed but the *CarOwnership* and *CarBrand* variables have a value `-1` to represent a non-applicable value (these folks do not have cars). If we wanted to re-code these values as `NA` we can simply subset the data for these elements and reassign the values.

```
customer$CarOwnership[customer$CarOwnership == "-1"] <- NA
customer$CarBrand[customer$CarBrand == "-1"] <- NA
```

Now if we check how many missing values are in each column we see *CarOwnership* and *CarBrand* included in the list:

```
missing <- colSums(is.na(customer))
sort(missing[missing > 0], decreasing = TRUE)
##  CarOwnership      CarBrand   NumberBirds        Gender   JobCategory
##           497           497            34            33            15
##     HomeOwner HouseholdSize    NumberDogs    NumberCats    NumberPets
##            13             8             8             7             6
```

We can also use this same approach to impute new non-missing values for any of these variables. For example, if we wanted to assign the median house hold size to those customers where this data is missing we could do so. The median household size across all customers is:

```r
median(customer$HouseholdSize, na.rm = TRUE)
## [1] 2
```

Thus, we can subset this variable for all those observations where the data is missing and add this value as the imputed value:

```r
# how many missing values exist in this variable
sum(is.na(customer$HouseholdSize))
## [1] 8

# impute these missing values with the median value
customer$HouseholdSize[is.na(customer$HouseholdSize)] <- median(customer$HouseholdSize, na.rm = TRUE)

# now how many missing values exist in this variable
sum(is.na(customer$HouseholdSize))
## [1] 0
```

Excluding missing values

Observations with missing values can provide interesting insights; however, sometimes we desire to simply remove all observations with missing values. We can exclude missing values by using `na.omit()` to omit all rows containing missing values.

```r
# remove all rows with any missing values
complete_cases <- na.omit(customer)

# no missing values
sum(is.na(complete_cases))
## [1] 0
```

Exercises

1. Install and load the `nycflights13` package. View the `flights` data that is built into this package.
2. What are the dimensions of this data set? What are the names of all the variables?
3. How many missing values are in the `flights` data? Omit all rows with missing values in them? Now how many observations does your data have?
4. Filter this data for only those flights that occurred on December 25th.
5. Select the following variables: *carrier*, *dep_delay*, *arr_delay*.
6. Create a new variable titled "recoup" that subtracts *dep_delay* from *arr_delay* to signal how much time in the air the flight recouped.
7. Compute the mean *recoup* value by *carrier*
8. Arrange the summarized *recoup* values to see which carrier tended to recoup the most (or lose the least amount of time while in the air).
9. Now perform #4-8 above in one step by combining them with the pipe (`%>%`) operator.

Exploratory Data Analysis

This module will show you how to systematically explore your data, a task that statisticians call exploratory data analysis, or EDA for short. EDA is an important part of any analytics project and should be performed prior to applying more sophisticated modeling techniques to ensure you have a firm understanding of your data. Moreover, many fundamental, day-to-day questions that organizations are asking can often be found with simple EDA approaches.

EDA is an iterative cycle where you:

- Generate questions about your data.
- Search for answers by visualizing, transforming, and describing your data.
- Use what you learn to refine your questions and/or generate new questions.

Your goal during EDA is to develop an understanding of your data. The easiest way to do this is to use questions as tools to guide your investigation and then use the skills you've learned throughout this guide to answer these questions. Fortunately, there is no definitive process to perform EDA; ultimately, it is a creative process with no hard rules. However, this module will illustrate several approaches to understand common attributes of your data.

Throughout this chapter we'll perform EDA on the *CustomerData_Merrimack.xlsx* data. Although not comprehensive, this case study will exemplify the entire process to include:

- Compute descriptive statistics
- Visualize your data
- Perform basic statistical inference
- Identify and measure relationships between variables

Prerequisites

In this module we'll use several packages that provide fundamental EDA capabilities. The tidyverse is a collection of R packages that share common philosophies and are designed to work together and simplify the EDA process. Let's install and load this package along with a few others we'll use:

```
install.packages("tidyverse")
library(tidyverse)
library(readxl)
```

Let's go ahead and read in our *CustomerData_Merrimack.xlsx* data and remove observations with missing values:

```
customer <- read_excel("data/CustomerData_Merrimack.xlsx", sheet = 1) %>%
  na.omit()
```

Descriptive Statistics

Descriptive statistics are the first pieces of information used to understand and represent a data set. There goal, in essence, is to describe the main features of numerical and categorical information with simple summaries. These summaries can be presented with a single numeric measure, frequency distributions, using summary tables, and many other ways. Here, I illustrate the most common forms of descriptive statistics for numerical and categorical data data but keep in mind there are numerous ways to describe and illustrate key features of data.

Numerical Data

- Central tendencies
- Variability
- Outliers
- Putting it together with `dplyr` functions

Central Tendency

There are three common measures of central tendency, all of which try to answer the basic question of which value is the most "typical." These are the mean (average of all observations), median (middle observation), and mode (appears most often). Each of these measures can be calculated for an individual variable or across all variables in a particular data frame. For example, if we are interested in finding the central tendency measures for the *CardSpendMonth* variable we can employ the following:

```
mean(customer$CardSpendMonth)
## [1] 3381.82
median(customer$CardSpendMonth)
## [1] 2775.2
```

Unfortunately, there is not a built-in function to compute the mode of a variable[^mode]. However, we can create a function that takes the vector as an input and gives the mode value as an output. Later on we'll see easier ways to identify most common values (and value ranges).

```
get_mode <- function(v) {
  unique_value <- unique(v)
  unique_value[which.max(tabulate(match(v, unique_value)))]
}

get_mode(customer$CardSpendMonth)
## [1] 0
```

Variability

The central tendencies give you a sense of the most typical values (prices in this case) but do not provide you with information on the variability of the values. Variability can be summarized in different ways, each providing you unique understanding of how the values are spread out.

Range

The range is a fairly crude measure of variability, defining the maximum and minimum values and the difference thereof. We can compute range summaries with the following:

```
# get the minimum value
min(customer$CardSpendMonth)
## [1] 0

# get the maximum value
max(customer$CardSpendMonth)
## [1] 39264.1

# get both the min and max values
range(customer$CardSpendMonth)
## [1]     0.0 39264.1
```

```
# compute the spread between min & max values
max(customer$CardSpendMonth) - min(customer$CardSpendMonth)
## [1] 39264.1
```

Percentiles

Given a certain percentage such as 25%, what is the dollar value such that 25% of cardholders are below it? This type of question leads to percentiles and quartiles. Specifically, for any percentage p, the pth percentile is the value such that a percentage p of all values are less than it. Similarly, the first, second, and third quartiles are the percentiles corresponding to $p = 25\%$, $p = 50\%$, and $p = 75\%$. These three values divide the data into four groups, each with (approximately) a quarter of all observations. Note that the second quartile is equal to the median by definition. These measures are easily computed in R:

```
# fivenum() function provides min, 25%, 50% (median), 75%, and max
fivenum(customer$CardSpendMonth)
## [1]     0.0  1836.4  2775.2  4194.6 39264.1

# default quantile() percentiles are 0%, 25%, 50%, 75%, and 100%
# provides same output as fivenum()
quantile(customer$CardSpendMonth)
##      0%     25%     50%     75%    100%
##     0.0  1836.4  2775.2  4194.6 39264.1

# we can customize quantile() for specific percentiles
quantile(customer$CardSpendMonth, probs = seq(from = 0, to = 1, by = .1))
##       0%      10%      20%      30%      40%      50%      60%      70%
##     0.00  1225.40  1650.60  2006.18  2377.48  2775.20  3233.32  3840.04
##      80%      90%     100%
##  4629.72  6131.82 39264.10

# we can quickly compute the difference between the 1st and 3rd quantile
IQR(customer$CardSpendMonth)
## [1] 2358.2
```

An alternative approach is to use the `summary` function which is a generic R function used to produce min, 1st quantile, median, mean, 3rd quantile, and max summary measures. However, though we do not see a difference here, note that the 1st and 3rd quantiles produced by `summary` may differ from the 1st and 3rd quantiles produced by `fivenum` and the default `quantile`. The reason for this is due to the lack of universal agreement on how the 1st and 3rd quartiles should be calculated.[^quant] Eric Cai provided a good blog post that discusses this difference in the R functions.

```
summary(customer$CardSpendMonth)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0    1836    2775    3382    4195   39264
```

Variance

Although the range provides a crude measure of variability and percentiles/quartiles provide an understanding of divisions of the data, the most common measures to summarize variability are variance and its derivatives (standard deviation and mean/median absolute deviation). We can compute each of these as follows:

```
# variance
var(customer$CardSpendMonth)
## [1] 6076157
```

```r
# standard deviation
sd(customer$CardSpendMonth)
## [1] 2464.986

# mean absolute deviation
mad(customer$CardSpendMonth, center = mean(customer$CardSpendMonth))
## [1] 1984.341

# median absolute deviation - note that the center argument defaults to median
# so it does not need to be specified, although I do just be clear
mad(customer$CardSpendMonth, center = median(customer$CardSpendMonth))
## [1] 1635.901
```

Shape

Two additional measures of a distribution that you will hear occasionally include skewness and kurtosis. Skewness is a measure of symmetry for a distribution. Negative values represent a *left-skewed* distribution where there are more extreme values to the left causing the mean to be less than the median. Positive values represent a *right-skewed* distribution where there are more extreme values to the right causing the mean to be more than the median.

Kurtosis is a measure of peakedness for a distribution. Negative values indicate a flat (platykurtic) distribution, positive values indicate a peaked (leptokurtic) distribution, and a value near 3 indicates a normal (mesokurtic) distribution.

We can get both skewness and kurtosis values using the `moments` package. Here, since our skewness is positive we can tell monthly card expenditures is right-skewed and since the kurtosis is much greater than 3 this suggests a very peaked distribution.

```r
library(moments)

skewness(customer$CardSpendMonth)
## [1] 3.051774
kurtosis(customer$CardSpendMonth)
## [1] 24.34354
```

Outliers

Outliers in data can distort predictions and affect their accuracy. Consequently, its important to understand if outliers are present and, if so, which observations are considered outliers. The `outliers` package provides a number of useful functions to systematically extract outliers. The functions of most use are `outlier()` and `scores()`. The `outlier` function gets the most extreme observation from the mean. The `scores` function computes the normalized (*z*, *t*, *chisq*, etc.) score which you can use to find observation(s) that lie beyond a given value.

```r
library(outliers)

# gets most extreme right-tail observation
outlier(customer$CardSpendMonth)

# gets most extreme left-tail observation
outlier(customer$CardSpendMonth, opposite = TRUE)

# observations that are outliers based on z-scores greater than 2.58. In other
```

```
# words, these observations exceed 2.58 standard deviations from the mean.
z_scores <- scores(customer$CardSpendMonth, type = "z")
which(abs(z_scores) > 2.58)

# outliers based on values less than or greater than the "whiskers" on a
# boxplot (1.5 x IQR or more below 1st quartile or above 3rd quartile)
which(scores(customer$CardSpendMonth, type = "iqr", lim = 1.5))
```

How you deal with outliers is a topic worthy of its own chapter; however, if you want to simply remove an outlier or replace it with the sample mean or median then I recommend the `rm.outlier` function provided also by the `outliers` package.

Putting it together with `dplyr` functions

As we saw in the last module, we can use these functions inside our `dplyr` functions for easy exploratory data analysis. Here we compute the average monthly card expenditure and the 95% confidence interval around that value. Thus, our level of 95% certainty about the true mean for female card expenditures is within the interval between $3,134.70 and $3,318.10 assuming that the original random variable is normally distributed, and the samples are independent.

```
customer %>%
  group_by(Gender) %>%
  summarize(avg_spend = mean(CardSpendMonth),
            Std.Dev = sd(CardSpendMonth),
            n = n()) %>%
  mutate(lower = avg_spend - (qnorm(0.975) * Std.Dev / sqrt(n)),
         upper = avg_spend + (qnorm(0.975) * Std.Dev / sqrt(n)))
## # A tibble: 2 x 6
##   Gender avg_spend  Std.Dev     n    lower    upper
##    <chr>     <dbl>    <dbl> <int>    <dbl>    <dbl>
## 1 Female  3226.398 2322.492  2464 3134.695 3318.100
## 2   Male  3539.482 2592.547  2429 3436.381 3642.582
```

Categorical Data

- Frequency tables
- Proportions tables
- Marginal tables
- Simplifying with `dplyr` functions

Frequency Tables

To produce contingency tables which calculate counts for each combination of categorical variables we can use R's `table()` function. For instance, we may want to get the total count of observations that fall into each region category.

```
# counts for regions
table(customer$Region)
##
##    1    2    3    4    5
##  984  981  980  944 1004
```

If we want to understand the number of observations by region *and* gender we can produce a cross classification table:

```
# cross classification counts for regions by gender
table(customer$Gender, customer$Region)
##
##            1   2   3   4   5
##   Female 506 501 495 455 507
##   Male   478 480 485 489 497
```

There are also functions such as `ftable` that allows us to create three-plus dimensional contingency tables. In this case we assess the count of customers by marital status, gender, and region:

```
table1 <- table(customer$MaritalStatus ,customer$Gender, customer$Region)
ftable(table1)
##                     1   2   3   4   5
##
## Married   Female  214 222 233 233 284
##           Male    224 222 240 232 244
## Unmarried Female  292 279 262 222 223
##           Male    254 258 245 257 253
```

Proportions Tables

We can also produce contingency tables that present the proportions (percentages) of each category or combination of categories. To do this we simply feed the frequency tables produced by `table()` to the `prop.table()` function. The following reproduces the previous tables but calculates the proportions rather than counts:

```
# percentages of clients across regions
table(customer$Region) %>% prop.table()
##
##         1         2         3         4         5
## 0.2011036 0.2004905 0.2002861 0.1929287 0.2051911

# percentages of clients across regions & gender
table(customer$Gender, customer$Region) %>% prop.table()
##
##                    1          2          3          4          5
##   Female 0.10341304 0.10239117 0.10116493 0.09298999 0.10361741
##   Male   0.09769058 0.09809933 0.09912119 0.09993869 0.10157368

table1 <- table(customer$MaritalStatus ,customer$Gender, customer$Region) %>%
  prop.table()
ftable(table1)
##                             1          2          3          4          5
##
## Married   Female  0.04373595 0.04537094 0.04761905 0.04761905 0.05804210
##           Male    0.04577969 0.04537094 0.04904966 0.04741467 0.04986716
## Unmarried Female  0.05967709 0.05702023 0.05354588 0.04537094 0.04557531
##           Male    0.05191089 0.05272839 0.05007153 0.05252401 0.05170652
```

We can add `round()` after `prop.table()` to round our values to a specified decimal:

```
# percentages of clients across regions
table(customer$Region) %>%
  prop.table() %>%
  round(2)
##
```

```
##    1    2    3    4    5
## 0.20 0.20 0.20 0.19 0.21

# percentages of clients across regions & gender
table(customer$Gender, customer$Region) %>%
  prop.table() %>%
  round(2)
##
##             1    2    3    4    5
##   Female 0.10 0.10 0.10 0.09 0.10
##   Male   0.10 0.10 0.10 0.10 0.10

# percentages of clients across regions, gender & marital status
table(customer$MaritalStatus, customer$Gender, customer$Region) %>%
  prop.table() %>%
  round(2) %>%
  ftable()
##                      1    2    3    4    5
##
## Married   Female  0.04 0.05 0.05 0.05 0.06
##           Male    0.05 0.05 0.05 0.05 0.05
## Unmarried Female  0.06 0.06 0.05 0.05 0.05
##           Male    0.05 0.05 0.05 0.05 0.05
```

Margins Tables

Margins show the total counts or percentages across columns or rows in a contingency table. For instance, if we go back to the cross classification counts for gender by region:

```
table(customer$Gender, customer$Region)
##
##            1   2   3   4   5
##   Female 506 501 495 455 507
##   Male   478 480 485 489 497
```

We can compute and add the column and row margins with `addmargins()`

```
table(customer$Gender, customer$Region) %>%
  addmargins()
##
##             1    2    3    4    5  Sum
##   Female  506  501  495  455  507 2464
##   Male    478  480  485  489  497 2429
##   Sum     984  981  980  944 1004 4893
```

We may want to understand the marginal distribution by row or column. For example, we may want to understand what the distribution of customers is by gender for each region. In other words, in region 1 what percent of customers are female versus male. We can use the `prop.table` function we saw earlier and incorporate the `margin` argument. `margin = 2` will provide the distribution of gender for each region.

```
table(customer$Gender, customer$Region) %>%
  prop.table(margin = 2)
##
##                   1         2         3         4         5
##   Female 0.5142276 0.5107034 0.5051020 0.4819915 0.5049801
##   Male   0.4857724 0.4892966 0.4948980 0.5180085 0.4950199
```

Alternatively, `margin = 1` will provide the row-wise distribution (i.e. how females are distributed across regions).

```
table(customer$Gender, customer$Region) %>%
  prop.table(margin = 1)
##
##                  1         2         3         4         5
##   Female 0.2053571 0.2033279 0.2008929 0.1846591 0.2057630
##   Male   0.1967888 0.1976122 0.1996706 0.2013174 0.2046110
```

Simplifying with `dplyr` Functions

I find using `dplyr` to get counts at multiple categorical levels much easier. For instance, the following shows how to use the `count` function from `dplyr` to count the number of observations for each region.

```
customer %>% count(Region)
## # A tibble: 5 x 2
##   Region     n
##    <dbl> <int>
## 1      1   984
## 2      2   981
## 3      3   980
## 4      4   944
## 5      5  1004
```

We can add additional categories to `count` observations at multiple levels.

```
customer %>% count(Region, Gender)
## Source: local data frame [10 x 3]
## Groups: Region [?]
##
## # A tibble: 10 x 3
##    Region Gender     n
##     <dbl>  <chr> <int>
##  1      1 Female   506
##  2      1   Male   478
##  3      2 Female   501
##  4      2   Male   480
##  5      3 Female   495
##  6      3   Male   485
##  7      4 Female   455
##  8      4   Male   489
##  9      5 Female   507
## 10      5   Male   497
```

Although this doesn't create our normal-looking contingency table as `table` does, it does allow us to pipe follow-on functions more easily. For example, we can find what region and job category combinations have the largest number of observations. Here we see that our customers tend to be in sales more than any other job category and that is consistent across all regions.

```
customer %>%
  count(Region, JobCategory) %>%
  arrange(desc(n))
## Source: local data frame [30 x 3]
## Groups: Region [5]
##
```

```
## # A tibble: 30 x 3
##    Region  JobCategory     n
##     <dbl>        <chr> <int>
##  1      5        Sales   338
##  2      2        Sales   328
##  3      3        Sales   321
##  4      1        Sales   310
##  5      4        Sales   302
##  6      2 Professional   286
##  7      3 Professional   277
##  8      4 Professional   271
##  9      1 Professional   264
## 10      5 Professional   259
## # ... with 20 more rows
```

We can also use `mutate` to create a new variable called "percent" that represents the proportion.

```
# percentages of clients across regions
customer %>%
  count(Region) %>%
  mutate(percent = n / sum(n))
## # A tibble: 5 x 3
##   Region     n   percent
##    <dbl> <int>     <dbl>
## 1      1   984 0.2011036
## 2      2   981 0.2004905
## 3      3   980 0.2002861
## 4      4   944 0.1929287
## 5      5  1004 0.2051911

# percentages of clients across regions & gender
customer %>%
  count(Region, Gender) %>%
  mutate(percent = n / sum(n))
## Source: local data frame [10 x 4]
## Groups: Region [5]
##
## # A tibble: 10 x 4
##    Region Gender     n   percent
##     <dbl>  <chr> <int>     <dbl>
##  1      1 Female   506 0.5142276
##  2      1   Male   478 0.4857724
##  3      2 Female   501 0.5107034
##  4      2   Male   480 0.4892966
##  5      3 Female   495 0.5051020
##  6      3   Male   485 0.4948980
##  7      4 Female   455 0.4819915
##  8      4   Male   489 0.5180085
##  9      5 Female   507 0.5049801
## 10      5   Male   497 0.4950199
```

Note that the second set of code above computes the percentage of clients for a particular region by gender. For example, the first row is telling us that for region 1, 51% of the customers are *Female* whereas 49% are *Male*. If we want to understand the percent that each category combination is of *all* the data we just need to `ungroup` prior to our `mutate` function. The code that follows allows us to identify the fact that each gender

by region represents about 10% of the data. Thus, we can be confident that we have equal dispersion of clients across the gender categories across all the regions served.

```
# percentages by category combination
customer %>%
  count(Region, Gender) %>%
  ungroup() %>%
  mutate(percent = n / sum(n)) %>%
  arrange(desc(percent))
## # A tibble: 10 x 4
##    Region Gender     n    percent
##     <dbl>  <chr> <int>      <dbl>
##  1      5 Female   507 0.10361741
##  2      1 Female   506 0.10341304
##  3      2 Female   501 0.10239117
##  4      5   Male   497 0.10157368
##  5      3 Female   495 0.10116493
##  6      4   Male   489 0.09993869
##  7      3   Male   485 0.09912119
##  8      2   Male   480 0.09809933
##  9      1   Male   478 0.09769058
## 10      4 Female   455 0.09298999
```

Exercises

1. What is the mean, median, standard deviation, and 85th percentile for *HHIncome*?
2. What is the 95% confidence interval for male and female mean *HHIncome*?
3. For the *HHIncome* variable identify how many observations are beyond 3 standard deviations from the mean?
4. How many observations are in each *JobCategory*?
5. What is the percentage of customers that are in the Sales *JobCategory*?
6. What are the percentages of customers in the Sales *JobCategory* across each region? Are these proportions consistent across gender?

Visualizing Data

- Getting started with plotting in R
- Advanced plotting with `ggplot2`

Getting Started with Plotting in R

For quick data exploration, base R plotting functions can provide an expeditious and straightforward approach to understanding your data. These functions are installed by default in base R and do not require additional visualization packages to be installed. This straightforward chapter should teach you the basics, and give you a good idea of what you want to do next.

In addition, I'll show how to make similar graphics with the `qplot()` function in `ggplot2`, which has a syntax similar to the base graphics functions. For each `qplot()` graph, there is also an equivalent using the more powerful `ggplot()` function which I illustrate in later visualization tutorials. This will, hopefully, help you transition to using `ggplot2` when you want to make more sophisticated graphics.

The following visualizations are covered:

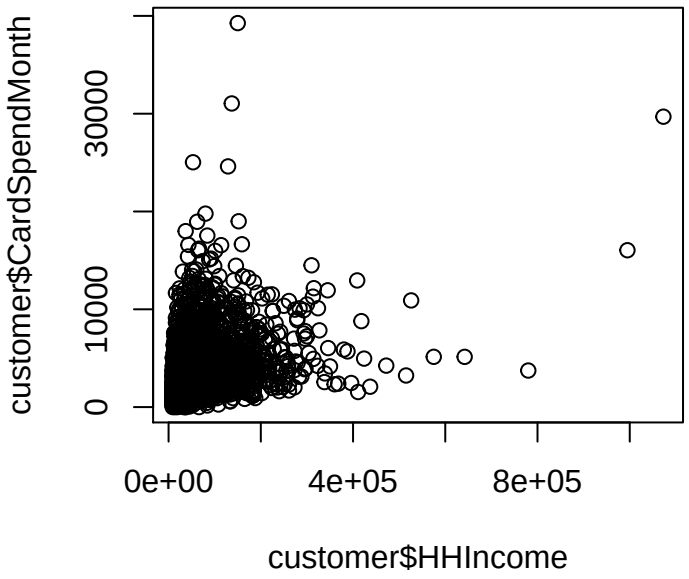
- Scatter plot

- Line chart
- Bar chart
- Histogram
- Box plot
- Stem & leaf plot

Scatter Plot

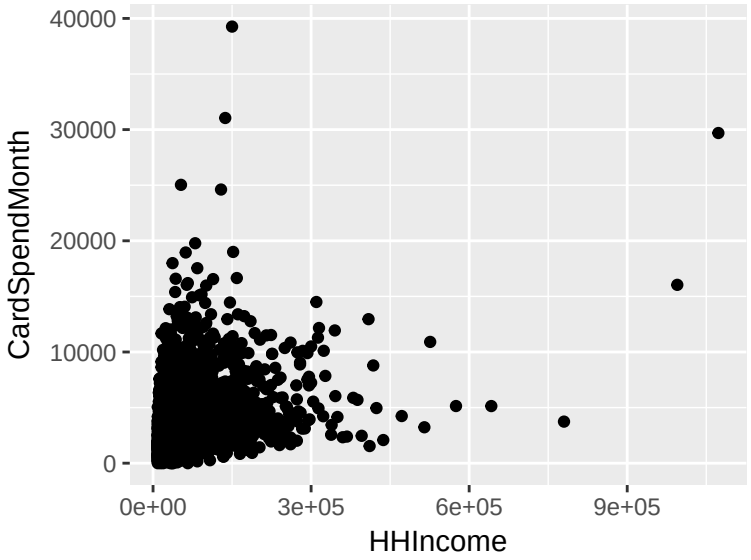
To make a scatter plot use `plot()` with a vector of x values and a vector of y values. Here we look at the relationship between household income and month card expenditures:

```
# base R
plot(x = customer$HHIncome, y = customer$CardSpendMonth)
```



You can get a similar result using `qplot()` from the `ggplot2` package. `ggplot2` was automatically loaded when you loaded the `tidyverse` package.

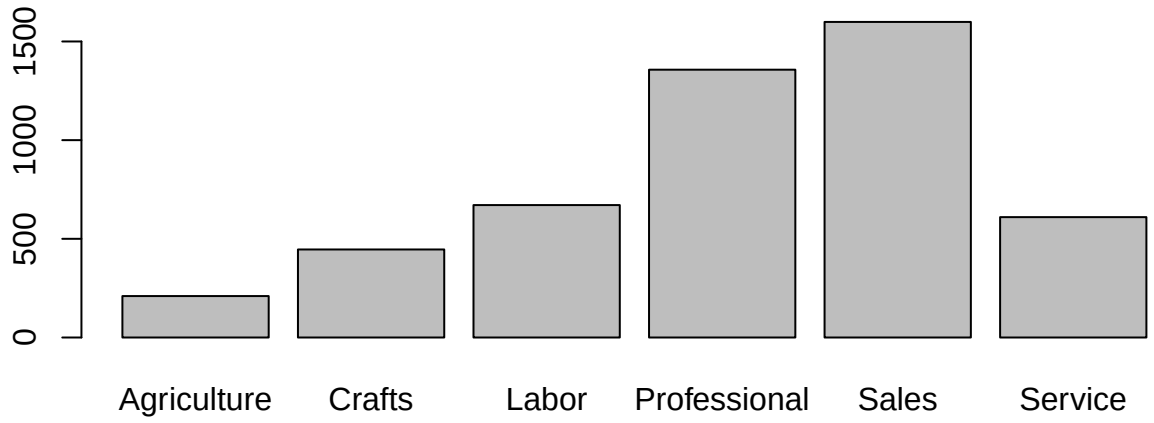
```
qplot(x = HHIncome, y = CardSpendMonth, data = customer)
```



Bar Chart

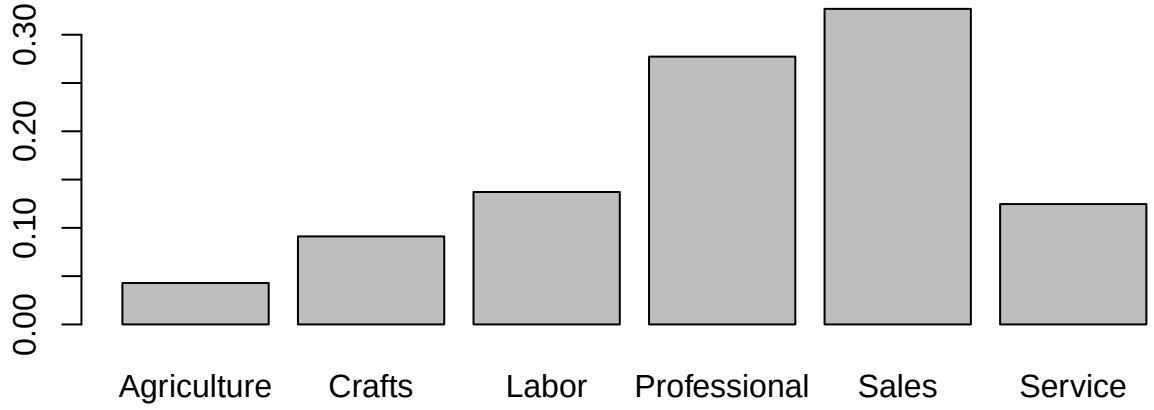
To make a bar chart of values, you first need to compute the counts for a categorical variable and then pass that to `barplot()`. For example we can visualize the frequency of job categories.

```
table(customer$JobCategory) %>%
  barplot()
```



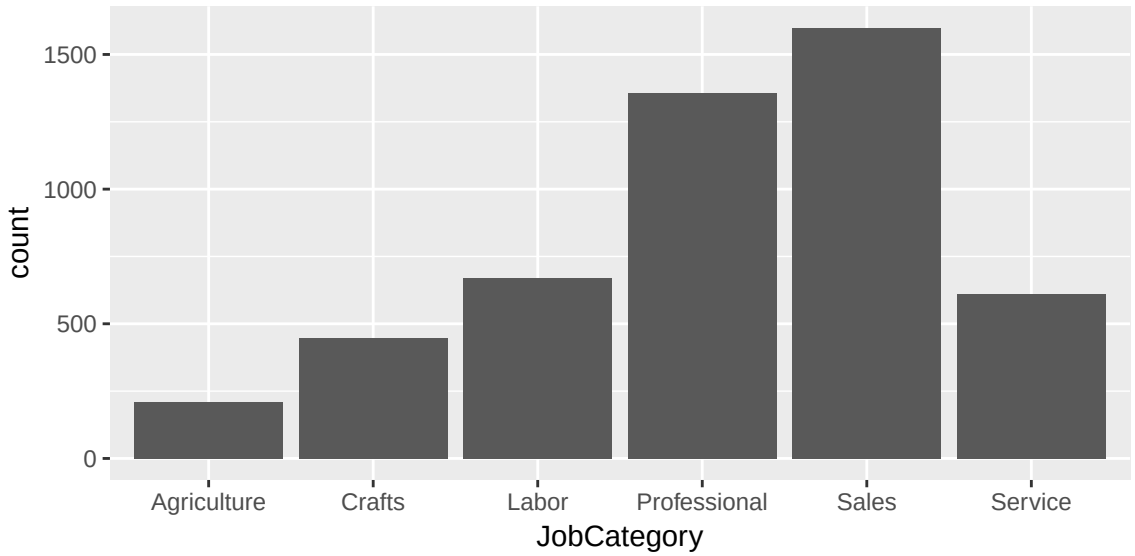
Similarly, we can convert this to a proportions chart by including `prop.table`:

```
table(customer$JobCategory) %>%
  prop.table() %>%
  barplot()
```



To get the same result using `qplot()` we use `geom = "bar"`. You'll learn more about "geoms" in the advanced plotting section.

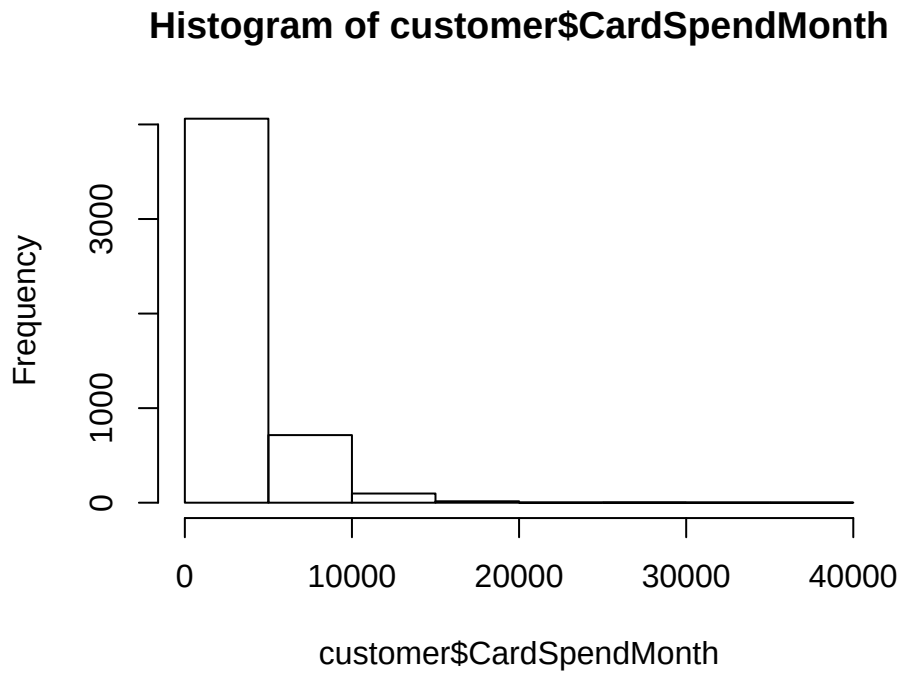
```
qplot(x= JobCategory, data = customer, geom = "bar")
```



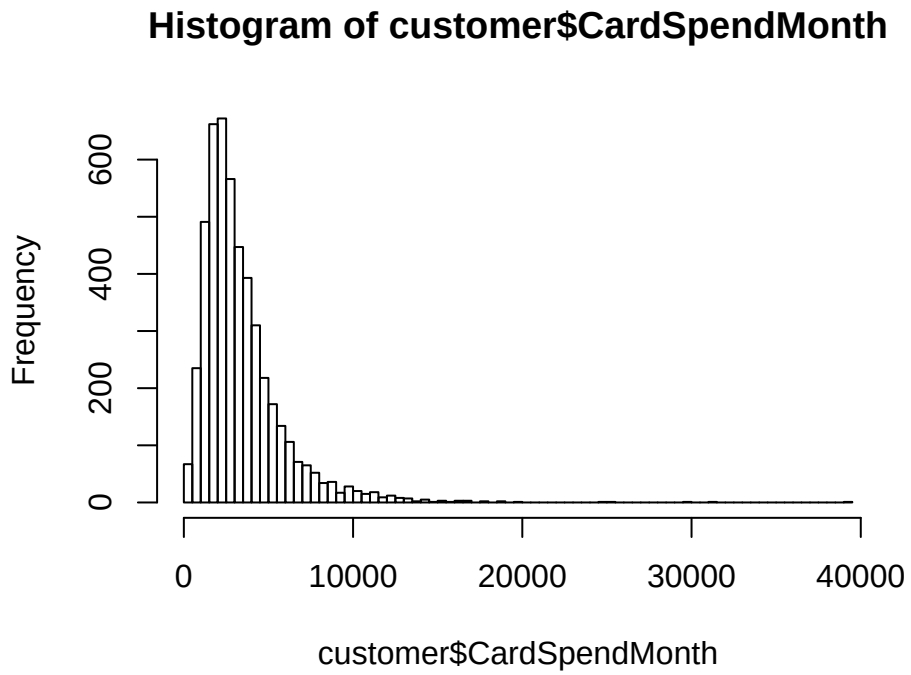
Histogram

To make a histogram, use `hist()` and pass it a single vector of values. You can also use the `breaks` argument to determine the size of the bins.

```
# default bins
hist(customer$CardSpendMonth)
```

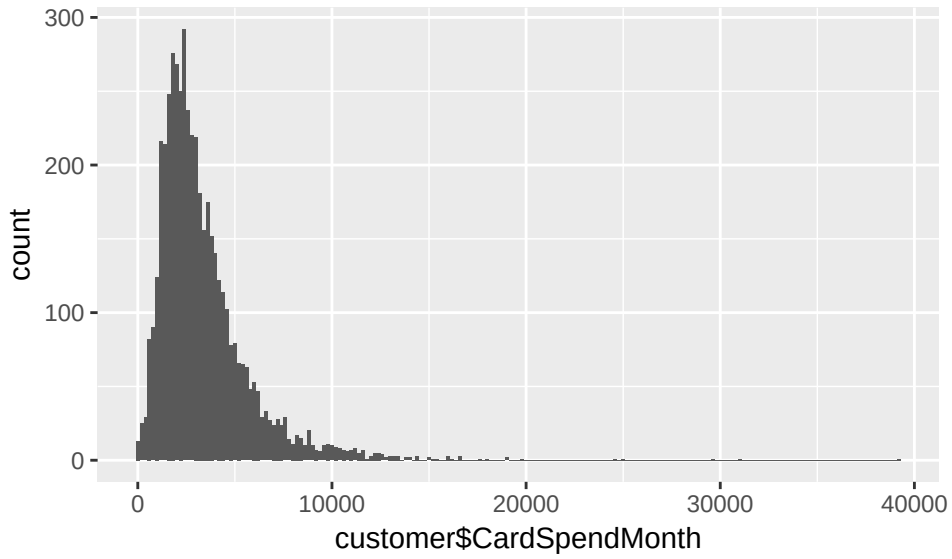



```
# adjust binning
hist(customer$CardSpendMonth, breaks = 100)
```



To get the same result using `qplot()` we don't need to specify a `geom` argument because when you feed `qplot()` with a single variable it will default to using a histogram. You can also control the binning by using the `binwidth` argument.

```
qplot(customer$CardSpendMonth, binwidth = 200)
```



Box Plot

Box plots are an alternative way to illustrate the distribution of a variable and is a concise way to illustrate the standard quantiles, shape, and outliers of data. As the generic diagram indicates, the box itself extends, left to right, from the 1st quartile to the 3rd quartile. This means that it contains the middle half of the data. The line inside the box is positioned at the median. The lines (whiskers) coming out either side of the box extend to 1.5 interquartile ranges (IQRs) from the quartiles. These generally include most of the data outside the box. More distant values, called outliers, are denoted separately by individual points.

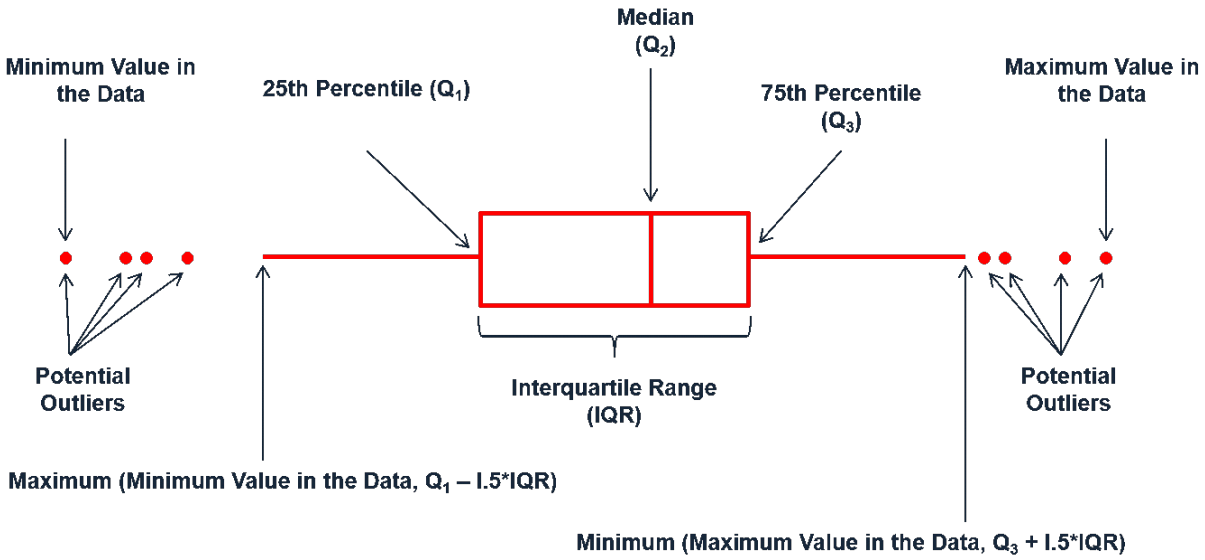
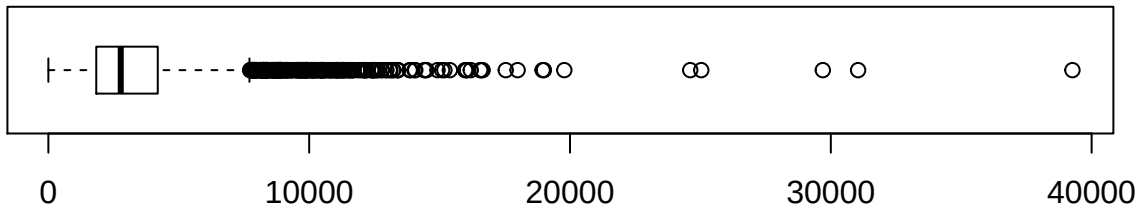


Figure 10: "Box Plot Description"

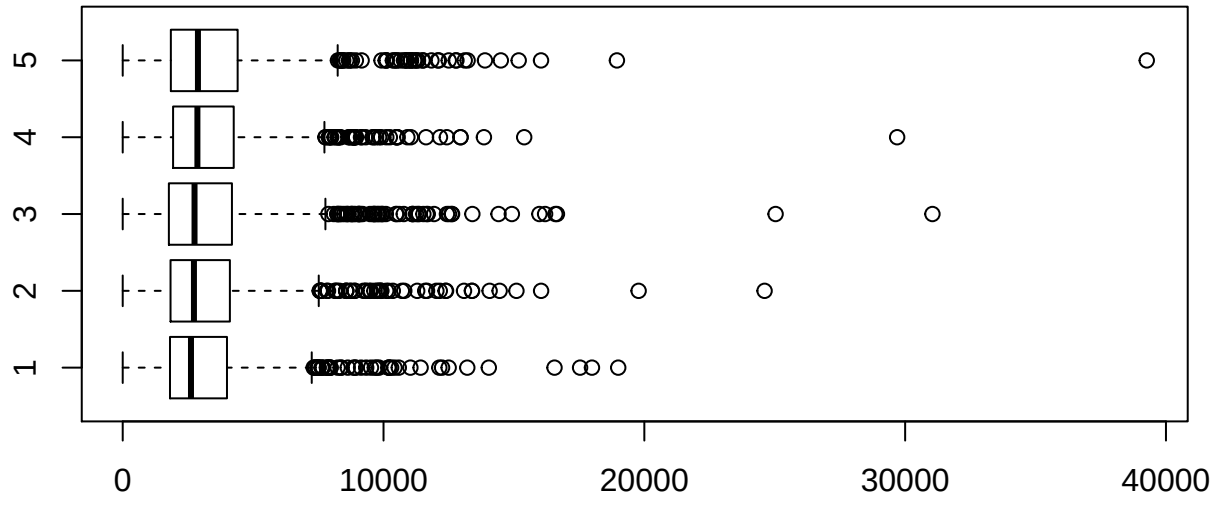
To make a box-whisker plot (aka box plot), use plot() and pass it x values that are categorical (aka factor) and a vector of y values. However, you need to ensure that the x values are factors otherwise you will get a scatter plot by default:

```
boxplot(customer$CardSpendMonth, horizontal = TRUE)
```



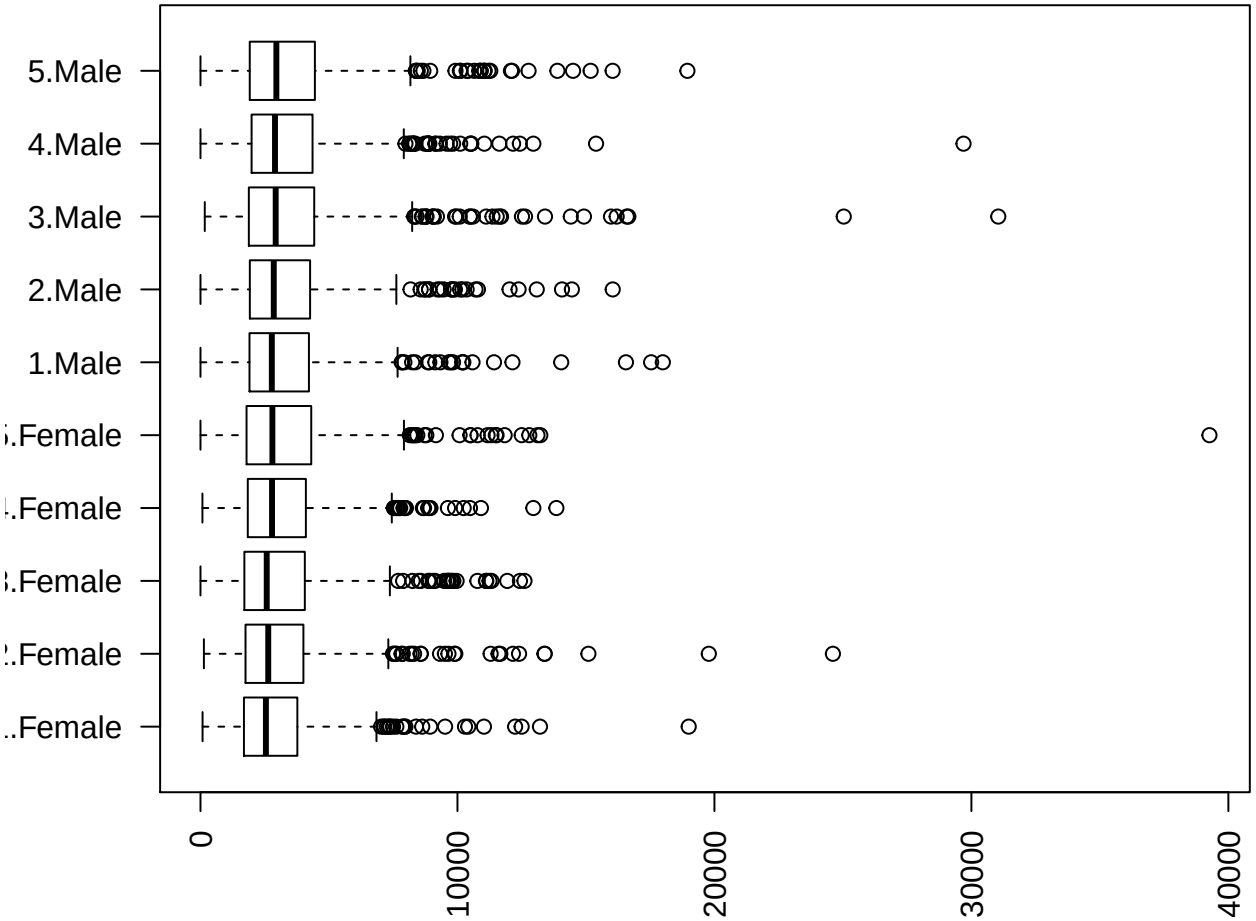
To get a box plot that displays the distribution of monthly expenditure values across the different regions we use the "~" to state that we want to assess y by x:

```
# boxplot of customer by cyl
boxplot(CardSpendMonth ~ Region, data = customer, horizontal = TRUE)
```



We can also assess interactions. In this case we look at the distribution of monthly expenditure values across the different regions and genders. Note that I use `las = 2` to rotate the axis labels.

```
# boxplot of customer based on interaction of two variables
boxplot(CardSpendMonth ~ Region + Gender, data = customer, horizontal = TRUE, las = 2)
```



Stem & Leaf Plot

To make a stem-and-leaf plot we can simply use the `stem()` function and pass it a vector of numeric values. A stem-and-leaf plot is primarily useful for smaller data sets. For example, we can see that the most common eruption times for Old Faithful is around 1.8 seconds and also around 4.4 seconds.

```
stem(faithful$eruptions)
##
##   The decimal point is 1 digit(s) to the left of the |
##
##   16 | 070355555588
##   18 | 000022233333335577777777888822335777888
##   20 | 00002223378800035778
##   22 | 0002335578023578
##   24 | 00228
##   26 | 23
##   28 | 080
##   30 | 7
```

```
##    32 | 2337
##    34 | 250077
##    36 | 0000823577
##    38 | 2333335582225577
##    40 | 000000335778888800223355557778
##    42 | 03335555778800233333555577778
##    44 | 022223355577800000000233333577788888
##    46 | 0000233357700000023578
##    48 | 00000022335800333
##    50 | 0370
```

Advanced Plotting with the `ggplot2` Package

Being able to create ***visualizations*** (graphical representations) of data is a key step in being able to communicate information and findings to others. In this chapter you will learn to use the ***`ggplot2`*** library to declaratively make beautiful plots or charts of your data. Although R does provide built-in plotting functions, the `ggplot2` library implements the Grammar of Graphics. This makes it particularly effective for describing how visualizations should represent data, and has turned it into the preeminent plotting library in R. Learning this library will allow you to make nearly any kind of (static) data visualization, customized to your exact specifications such as this re-creation of a classic visualiation provided in Edward Tufte's book Visual Display of Quantitative Information.

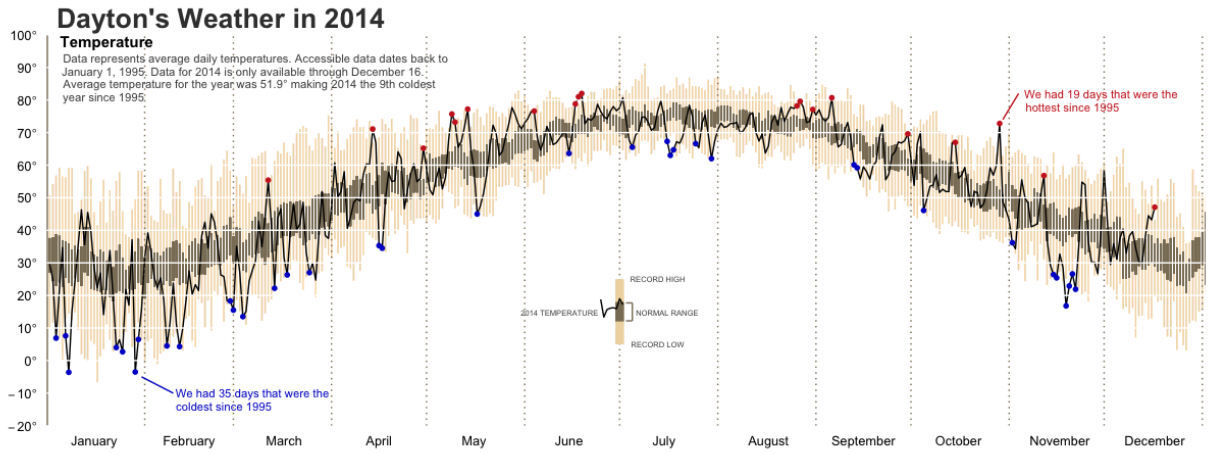


Figure 11: Re-created Tufte Illustration

This section will provide a general introduction to the ggplot syntax to include:[^adapted]

- Grammar of graphics: Grammar of graphics gives us a way to talk about parts of a plot
- The basics: Understanding the basics of the `ggplot` grammar
- Aesthetic mappings: Mapping variables to visualization characteristics
- Specifying geometric shapes: Plotting data with geometric shapes
- Managing scales: Understanding the different scales you can control
- Coordinate systems: Adjusting cartesian coordinate system
- Facets: Creating small multiples
- Labels & annotations: Ways to annotate your visualizations
- Additional resources on `ggplot2`: Resources to learn more about `ggplot2`
- Other visualization libraries: Popular interactive visualizations

Grammar of Graphics

Just as the grammar of language helps us construct meaningful sentences out of words, the ***Grammar of Graphics*** helps us to construct graphical figures out of different visual elements. This grammar gives us a way to talk about parts of a plot: all the circles, lines, arrows, and words that are combined into a diagram for visualizing data. Originally developed by Leland Wilkinson, the Grammar of Graphics was adapted by Hadley Wickham to describe the components of a plot, including

- the **data** being plotted
- the **geometric objects** (circles, lines, etc.) that appear on the plot
- a set of mappings from variables in the data to the **aesthetics** (appearance) of the geometric objects
- a **statistical transformation** used to calculate the data values used in the plot
- a **position adjustment** for locating each geometric object on the plot
- a **scale** (e.g., range of values) for each aesthetic mapping used
- a **coordinate system** used to organize the geometric objects
- the **facets** or groups of data shown in different plots

Wickham further organizes these components into **layers**, where each layer has a single *geometric object, statistical transformation, and position adjustment*. Following this grammar, you can think of each plot as a set of layers of images, where each image's appearance is based on some aspect of the data set.

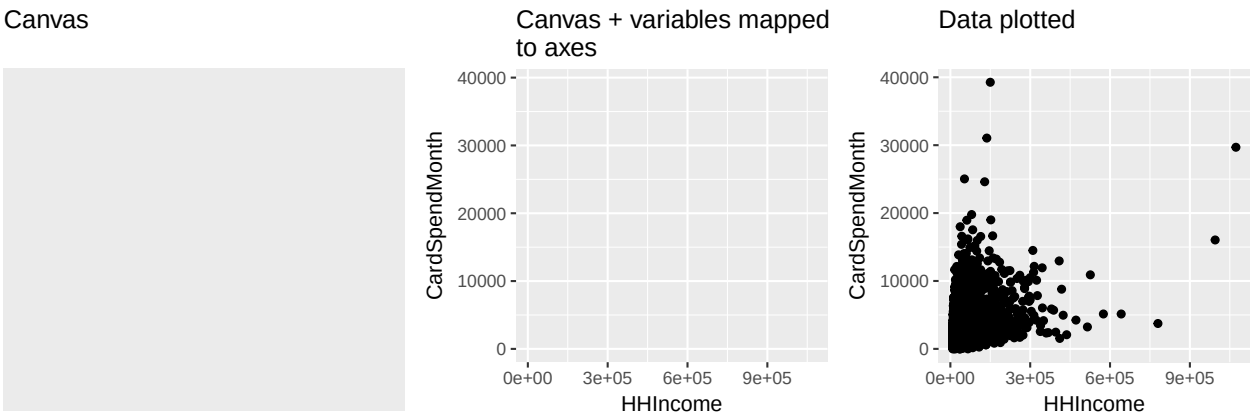
All together, this grammar enables us to discuss what plots look like using a standard set of vocabulary. And similar to how `dplyr` provides efficient data transformation and manipulation, `ggplot2` provides more efficient ways to create specific visual images.

The Basics

In order to create a plot, you:

1. Call the `ggplot()` function which creates a blank canvas
2. Specify **aesthetic mappings**, which specifies how you want to map variables to visual aspects. In this case we are simply mapping the *HHIncome* and *CardSpendMonth* variables to the x- and y-axes.
3. You then add new layers that are geometric objects which will show up on the plot. In this case we add `geom_point` to add a layer with *points* (dot) elements as the geometric shapes to represent the data.

```
# create canvas
ggplot(customer)

# variables of interest mapped
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth))

# data plotted
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point()
```



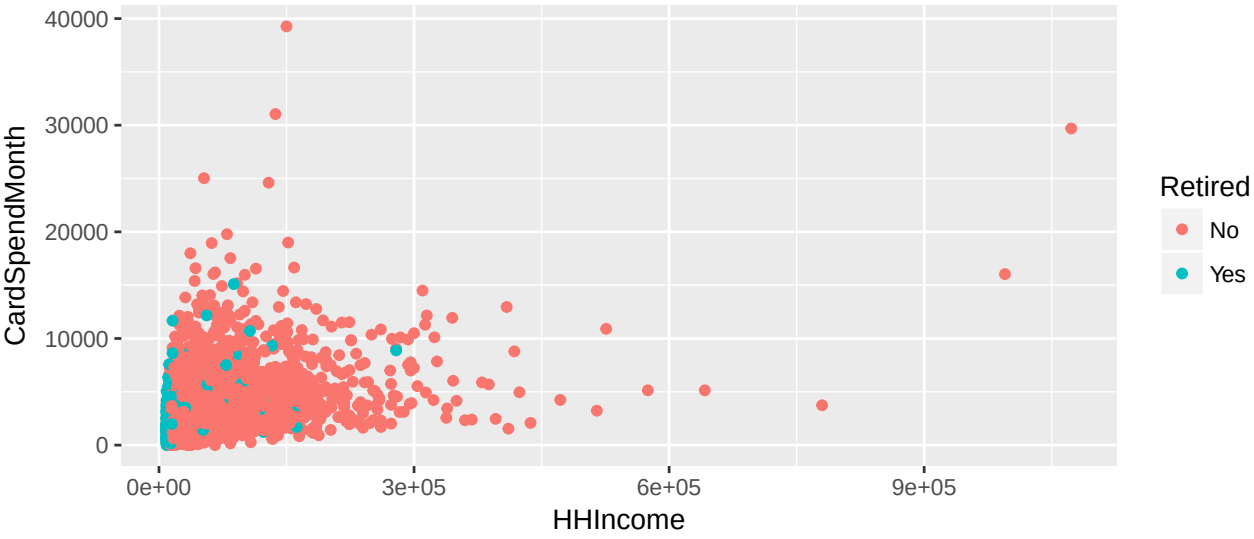
Note that when you added the `geom` layer you used the addition (`+`) operator. As you add new layers you will always use `+` to add onto your visualization.

Aesthetic Mappings

The ***aesthetic mappings*** take properties of the data and use them to influence visual characteristics, such as *position*, *color*, *size*, *shape*, or *transparency*. Each visual characteristic can thus encode an aspect of the data and be used to convey information.

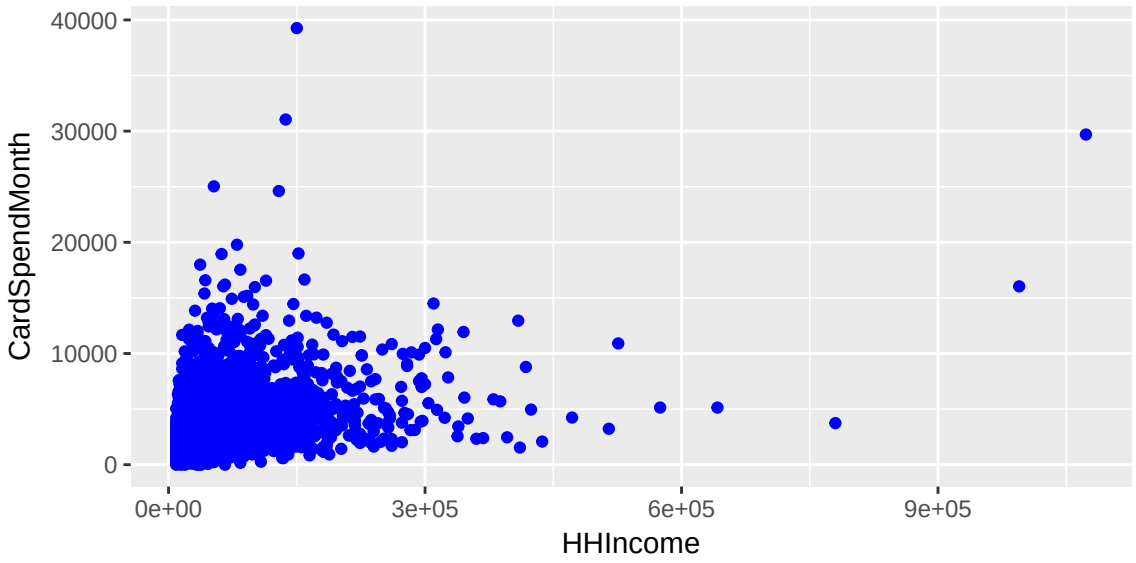
All aesthetics for a plot are specified in the aes() function call (later in this tutorial you will see that each `geom` layer can have its own `aes` specification). For example, we can add a mapping to a *color* characteristic to identify retired vs. non-retired customers:

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point()
```



Note that using the `aes()` function will cause the visual channel to be based on the data specified in the argument. For example, using `aes(color = "blue")` won't cause the geometry's color to be "blue", but will instead cause the visual channel to be mapped from the vector `c("blue")` — as if we only had a single type of engine that happened to be called "blue". If you wish to apply an aesthetic property to an entire geometry, you can set that property as an argument to the `geom` method, outside of the `aes()` call:

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point(color = "blue")
```



Specifying Geometric Shapes

Building on these basics, `ggplot2` can be used to build almost any kind of plot you may want. These plots are declared using functions that follow from the Grammar of Graphics.

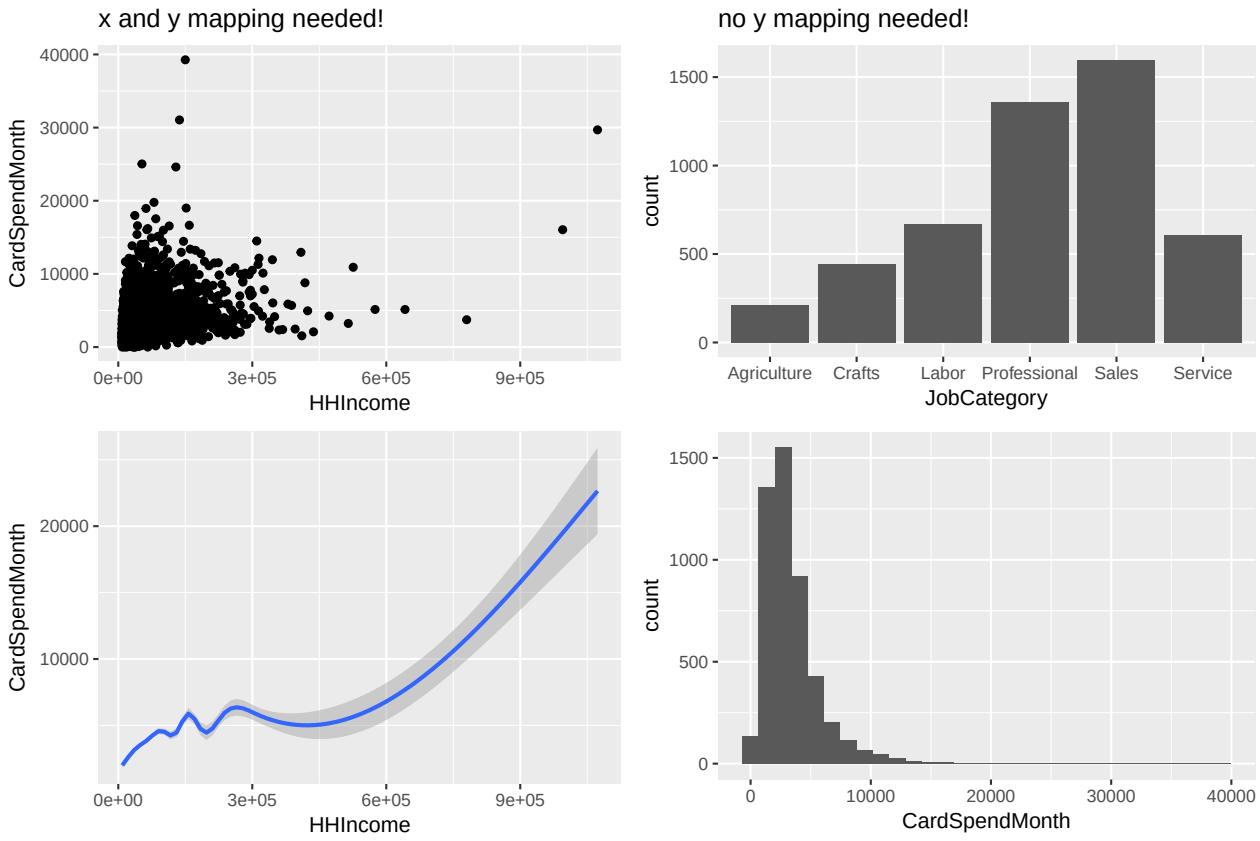
The most obvious distinction between plots is what **geometric objects** (`geoms`) they include. `ggplot2` supports a number of different types of `geoms`, including:

- `geom_point` for drawing individual points (e.g., a scatter plot)
- `geom_line` for drawing lines (e.g., for a line charts)
- `geom_smooth` for drawing smoothed lines (e.g., for simple trends or approximations)
- `geom_bar` for drawing bars (e.g., for bar charts)
- `geom_histogram` for drawing binned values (e.g. a histogram)
- `geom_polygon` for drawing arbitrary shapes
- `geom_map` for drawing polygons in the shape of a map! (You can access the data to use for these maps by using the `map_data()` function).

Each of these geometries will leverage the aesthetic mappings supplied although the specific visual properties that the data will map to will vary. For example, you can map data to the `shape` of a `geom_point` (e.g., if they should be circles or squares), or you can map data to the `linetype` of a `geom_line` (e.g., if it is solid or dotted), but not vice versa.

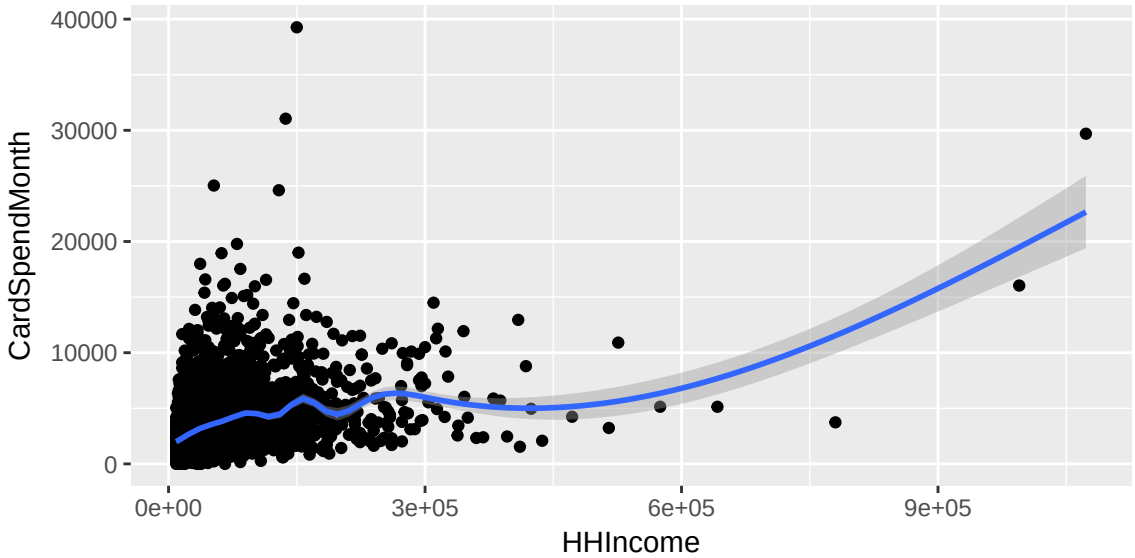
Almost all `geoms` require an `x` and `y` mapping at the bare minimum.

```
# Left column: x and y mapping needed!
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point()

ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_smooth()

# Right column: no y mapping needed!
ggplot(data = customer, aes(x = JobCategory)) +
  geom_bar()

ggplot(data = customer, aes(x = CardSpendMonth)) +
  geom_histogram()
```



What makes this really powerful is that you can add ***multiple*** geometries to a plot, thus allowing you to create complex graphics showing multiple aspects of your data.

```
# plot with both points and smoothed line
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point() +
  geom_smooth()
```

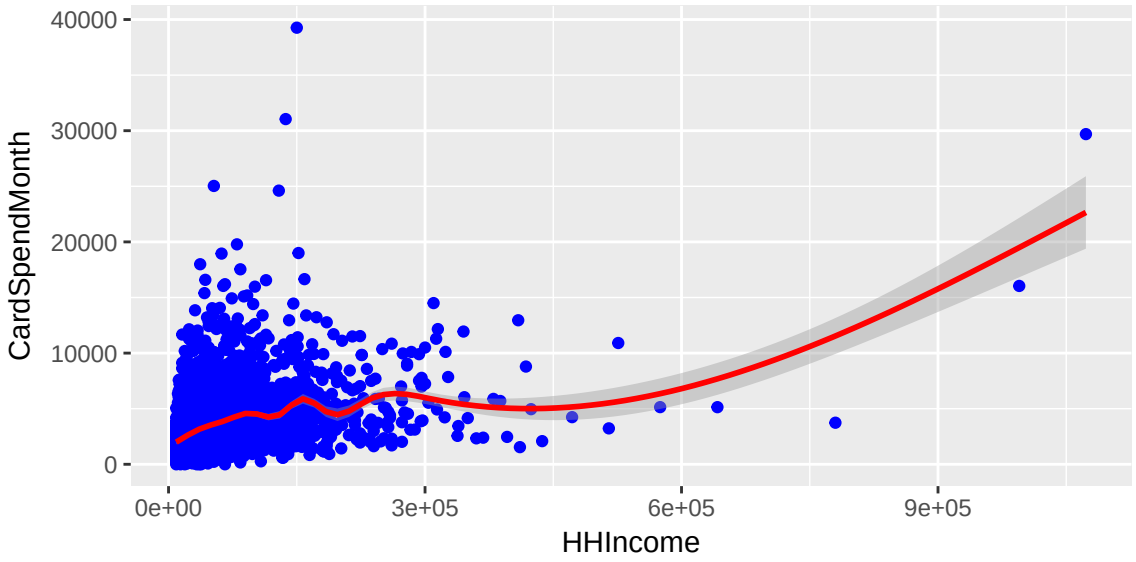


Of course the aesthetics for each `geom` can be different, so you could show multiple lines on the same plot (or

with different colors, styles, etc). It's also possible to give each `geom` a different data argument, so that you can show multiple data sets in the same plot.

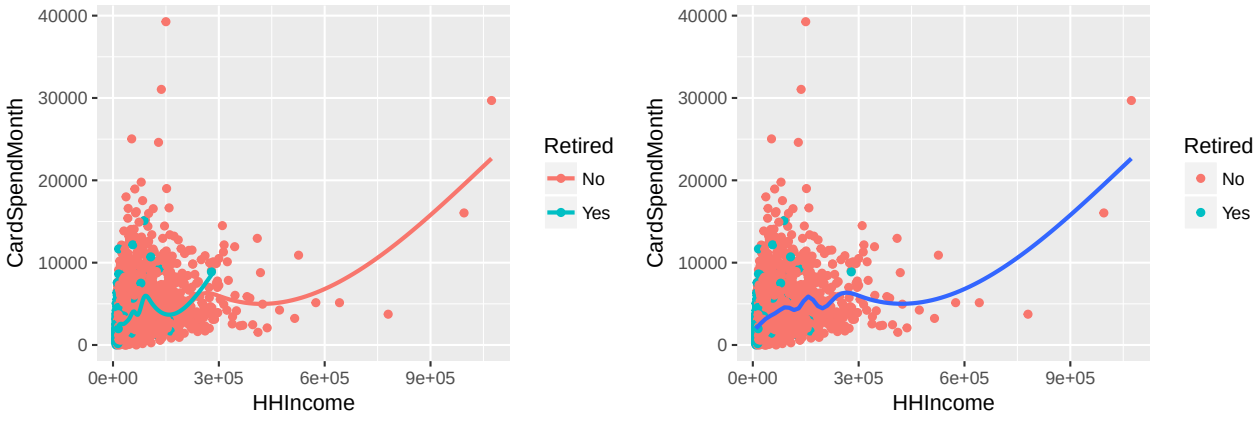
For example, we can plot both points and a smoothed line for the same `x` and `y` variable but specify unique colors within each `geom`:

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point(color = "blue") +
  geom_smooth(color = "red")
```



So as you can see if we specify an aesthetic within `ggplot` it will be passed on to each `geom` that follows. Or we can specify certain aes within each `geom`, which allows us to only show certain characteristics for that specify layer (i.e. `geom_point`).

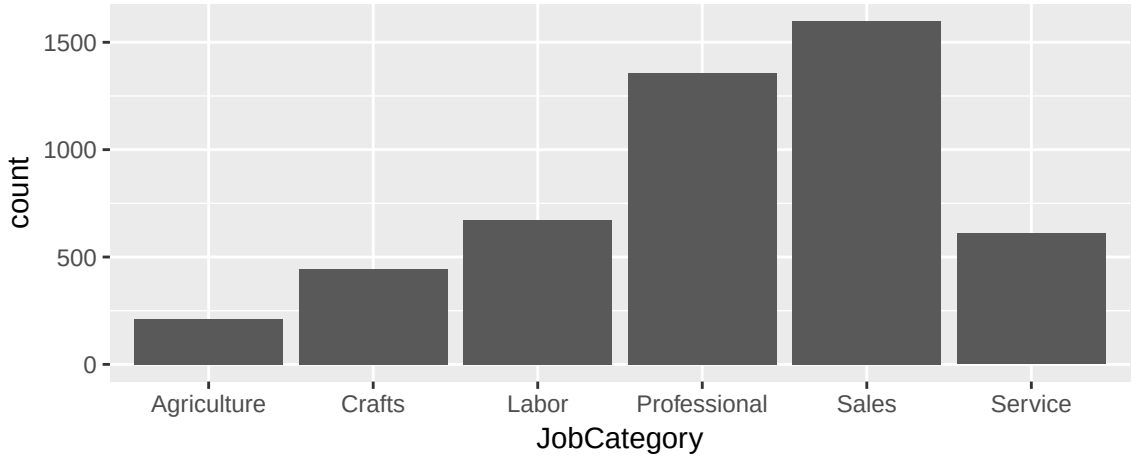
```
# color aesthetic passed to each geom layer
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point() +
  geom_smooth(se = FALSE)

# color aesthetic specified for only the geom_point layer
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point(aes(color = Retired)) +
  geom_smooth(se = FALSE)
```



Statistical Transformations

If you look at the below bar chart, you'll notice that the the y axis was defined for us as the *count* of elements that have the particular type. This count isn't part of the data set (it's not a column in our customer data), but is instead a ***statistical transformation*** that the `geom_bar` automatically applies to the data. In particular, it applies the `stat_count` transformation.

```
ggplot(customer, aes(x = JobCategory)) +
  geom_bar()
```

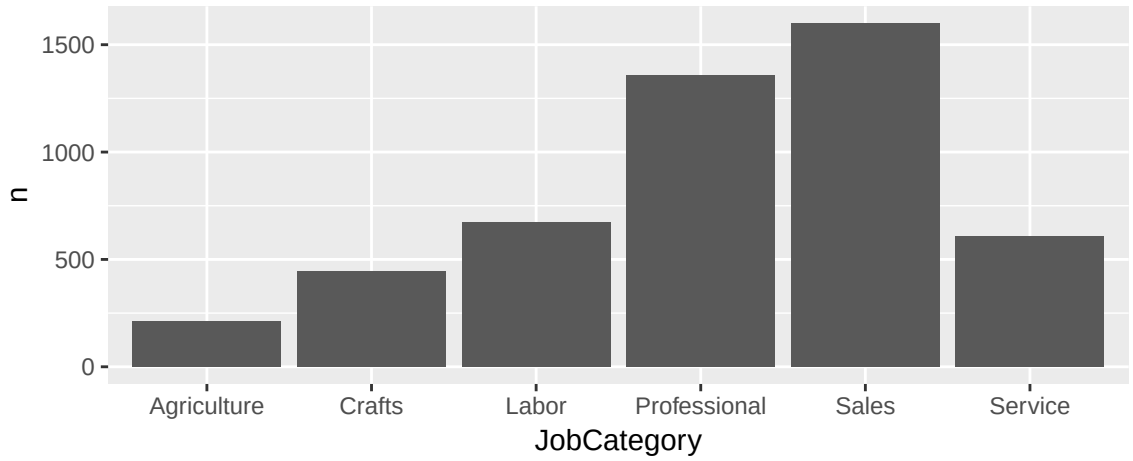


`ggplot2` supports many different statistical transformations. For example, the "identity" transformation will leave the data "as is". You can specify which statistical transformation a `geom` uses by passing it as the `stat` argument. For example, consider our data already had the count as a variable:

```
Region_count <- dplyr::count(customer, JobCategory)
Region_count
## # A tibble: 6 x 2
##    JobCategory     n
##          <chr> <int>
## 1  Agriculture   210
## 2       Crafts   446
## 3        Labor   671
## 4 Professional  1357
## 5        Sales  1599
## 6      Service   610
```

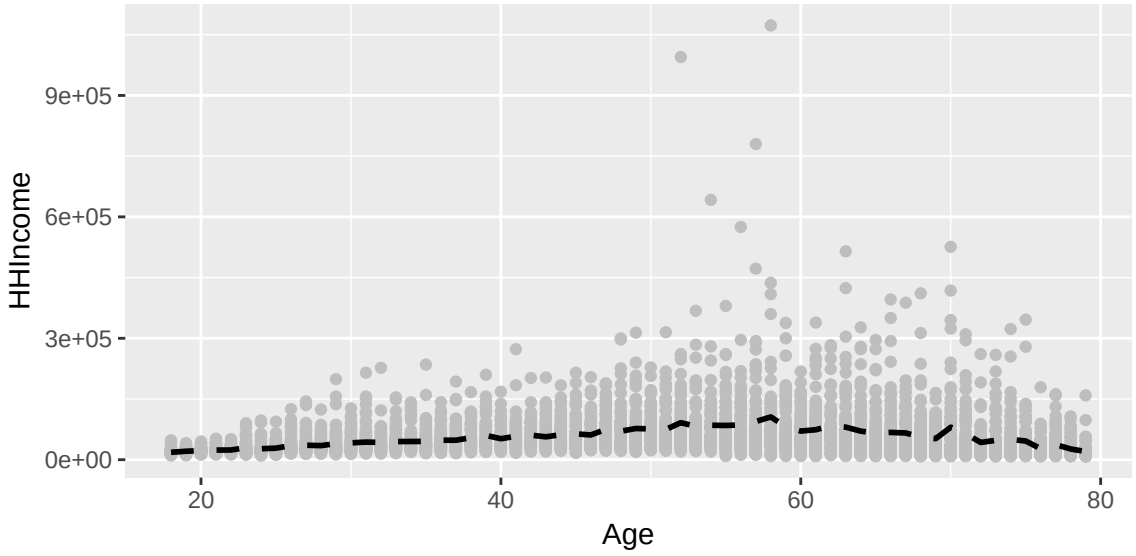
We can use `stat = "identity"` within `geom_bar` to plot our bar height values to this variable. Also, note that we now include n for our y variable:

```
ggplot(Region_count, aes(x = JobCategory, y = n)) +
  geom_bar(stat = "identity")
```



We can also call `stat_` functions directly to add additional layers. For example, here we create a scatter plot of HHIncome for each Age group and then use `stat_summary` to plot the mean HHIncome at each Age group.

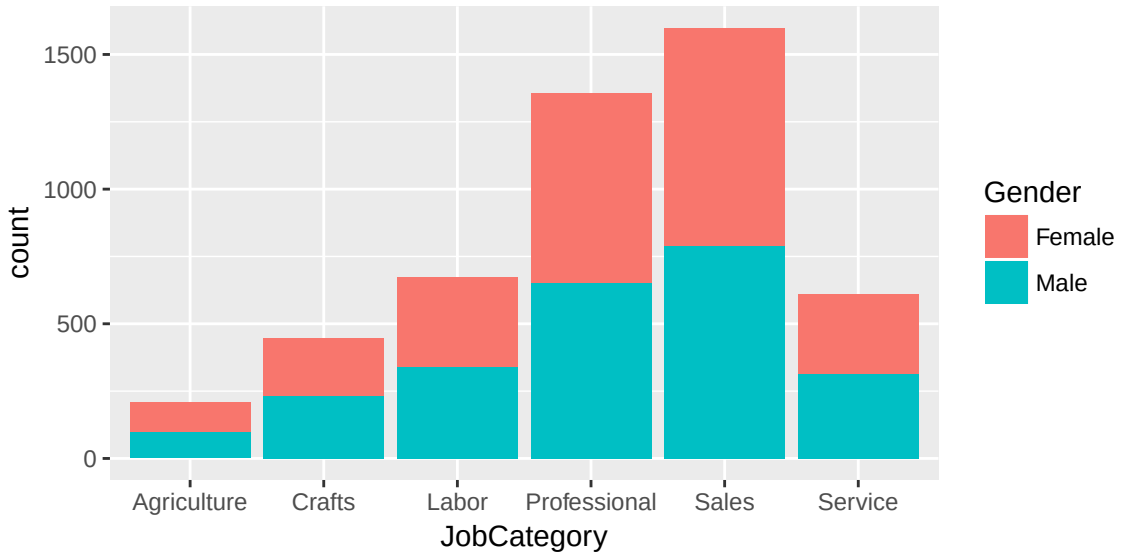
```
ggplot(customer, aes(Age, HHIncome)) +
  geom_point(color = "grey") +
  stat_summary(fun.y = "mean", geom = "line", size = 1, linetype = "dashed")
```



Position Adjustments

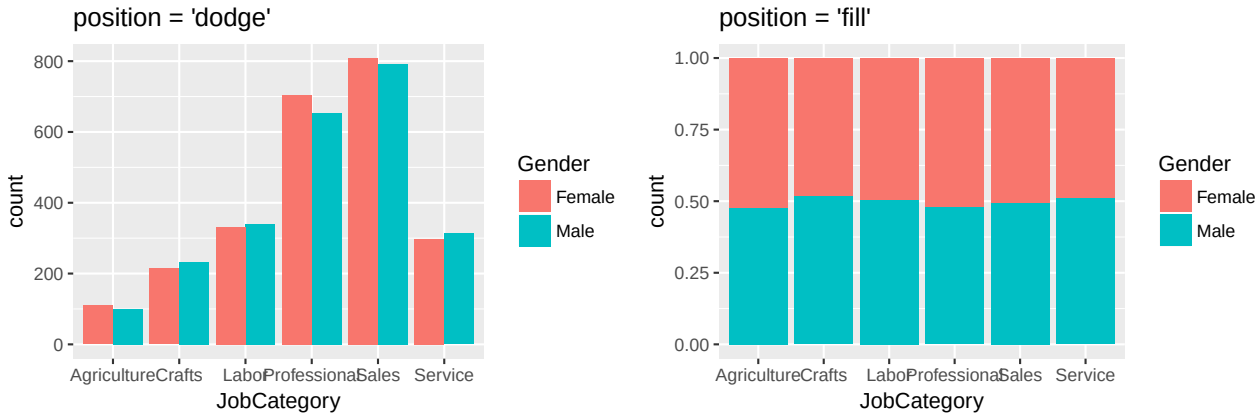
In addition to a default statistical transformation, each `geom` also has a default ***position adjustment*** which specifies a set of "rules" as to how different components should be positioned relative to each other. This position is noticeable in a `geom_bar` if you map a different variable to the color visual characteristic:

```
# bar chart of job category, colored by gender
ggplot(customer, aes(x = JobCategory, fill = Gender)) +
  geom_bar()
```



The `geom_bar` by default uses a position adjustment of `"stack"`, which makes each rectangle's height proportional to its value and stacks them on top of each other. We can use the `position` argument to specify what position adjustment rules to follow:

```
# position = "dodge": values next to each other
ggplot(customer, aes(x = JobCategory, fill = Gender)) +
  geom_bar(position = "dodge")

# position = "fill": percentage chart
ggplot(customer, aes(x = JobCategory, fill = Gender)) +
  geom_bar(position = "fill")
```



Check the documentation for each particular geom to learn more about its positioning adjustments.

Managing Scales

Whenever you specify an aesthetic mapping, `ggplot` uses a particular ***scale*** to determine the range of values that the data should map to. Thus when you specify

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point()
```

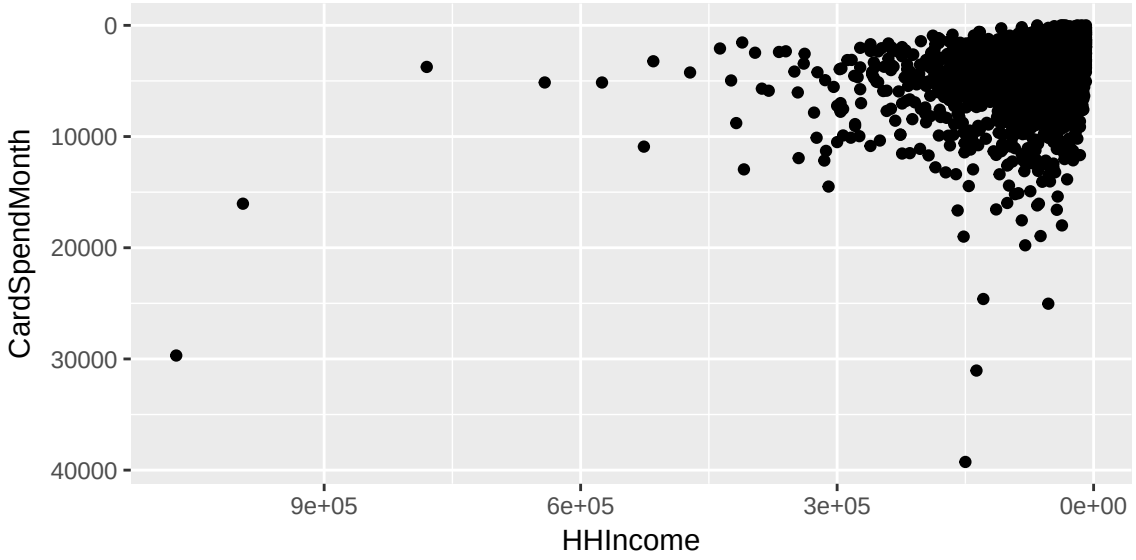
`ggplot` automatically adds a scale for each mapping to the plot:

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point() +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_colour_discrete()
```

Each scale can be represented by a function with the following name: `scale_`, followed by the name of the aesthetic property, followed by an `_` and the name of the scale. A `continuous` scale will handle things like numeric data (where there is a *continuous* set of numbers), whereas a `discrete` scale will handle things like colors (since there is a small list of *distinct* colors).

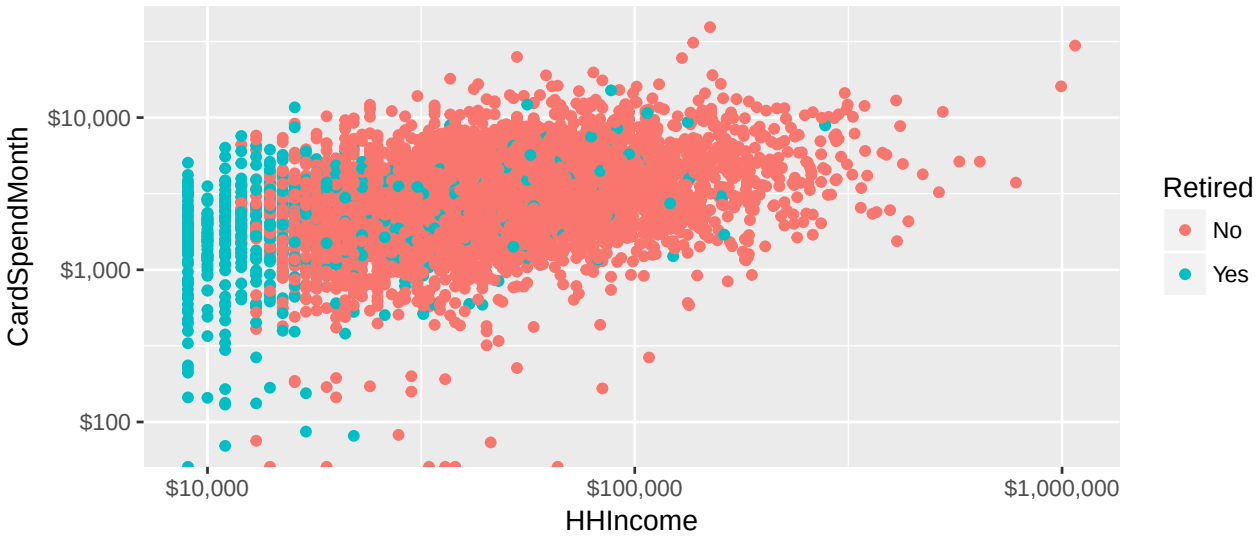
While the default scales will work fine, it is possible to explicitly add different scales to replace the defaults. For example, you can use a scale to change the direction of an axis:

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point() +
  scale_x_reverse() +
  scale_y_reverse()
```



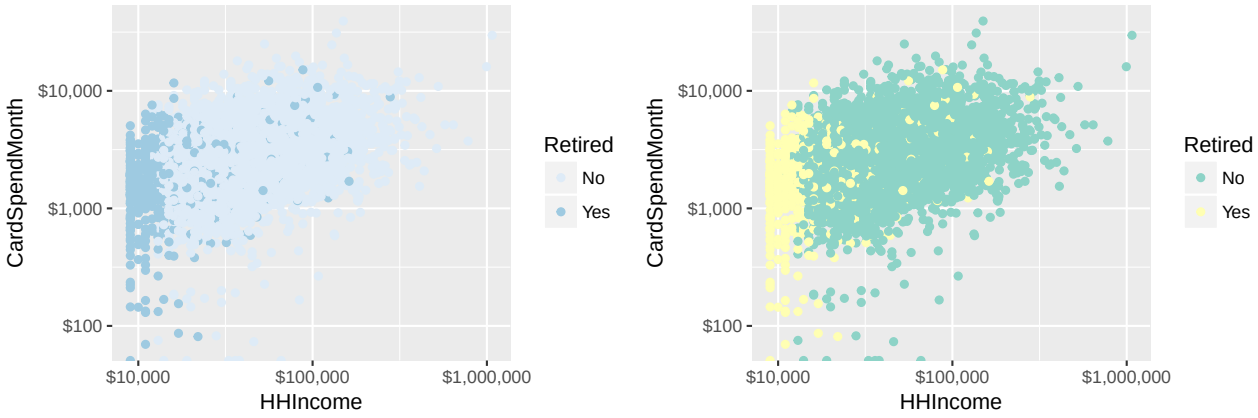
Similarly, you can use `scale_x_log10()` and `scale_x_sqrt()` to transform your scale. You can also use `scales` to format your axes:

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point() +
  scale_x_log10(labels = scales::dollar) +
  scale_y_log10(labels = scales::dollar)
```



A common parameter to change is which set of colors to use in a plot. While you can use the default coloring, a more common option is to leverage the pre-defined palettes from colorbrewer.org. These color sets have been carefully designed to look good and to be view-able to people with certain forms of color blindness. We can leverage color brewer palettes by specifying the `scale_color_brewer()` function, passing the palette as an argument.

```
# default color brewer
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point() +
  scale_x_log10(labels = scales::dollar) +
  scale_y_log10(labels = scales::dollar) +
  scale_color_brewer()

# specifying color palette
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point() +
  scale_x_log10(labels = scales::dollar) +
  scale_y_log10(labels = scales::dollar) +
  scale_color_brewer(palette = "Set3")
```



Note that you can get the palette name from the colorbrewer website by looking at the scheme query parameter in the URL. Or see the diagram here and hover the mouse over each palette for the name.

You can also specify *continuous* color values by using a gradient scale, or manually specify the colors you

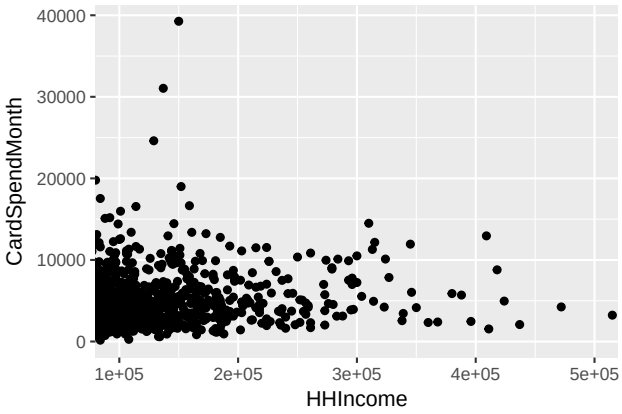
want to use as a named vector.

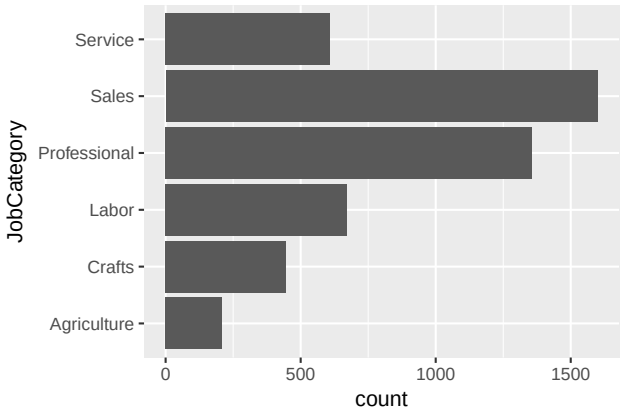
Coordinate Systems

The next term from the Grammar of Graphics that can be specified is the ***coordinate system***. As with scales, coordinate systems are specified with functions that all start with `coord_` and are added as a layer. There are a number of different possible coordinate systems to use, including:

- `coord_cartesian` the default cartesian coordinate system, where you specify x and y values (e.g. allows you to zoom in or out).
- `coord_flip` a cartesian system with the x and y flipped
- `coord_fixed` a cartesian system with a "fixed" aspect ratio (e.g., 1.78 for a "widescreen" plot)
- `coord_polar` a plot using polar coordinates
- `coord_quickmap` a coordinate system that approximates a good aspect ratio for maps. See documentation for more details.

```
# zoom in with coord_cartesian
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point() +
  coord_cartesian(xlim = c(500000, 100000))

# flip x and y axis with coord_flip
ggplot(customer, aes(x = JobCategory)) +
  geom_bar() +
  coord_flip()
```

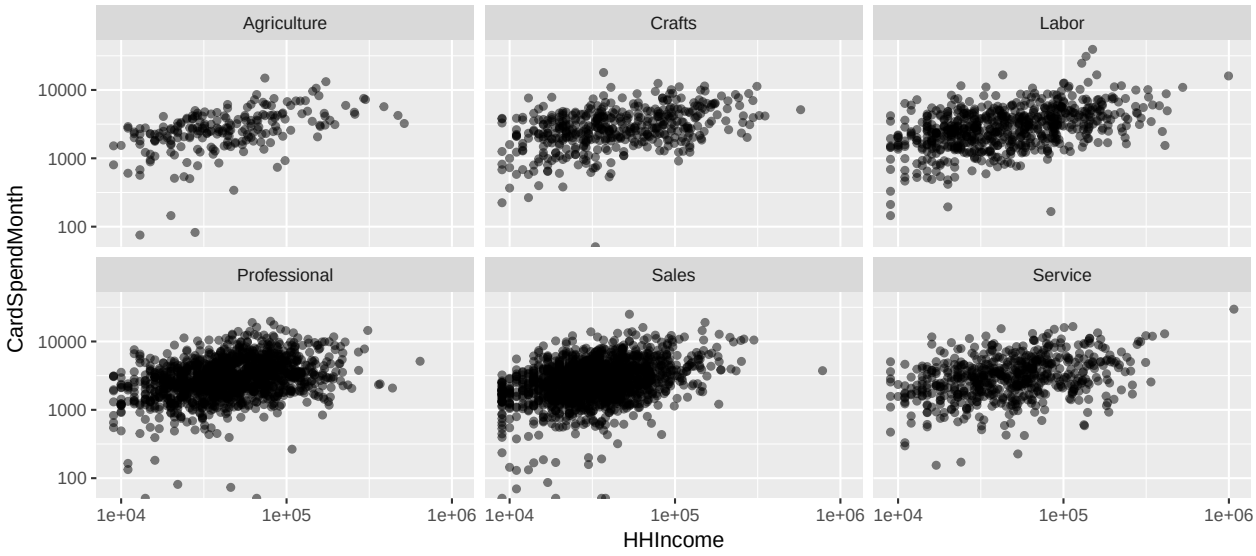




Facets

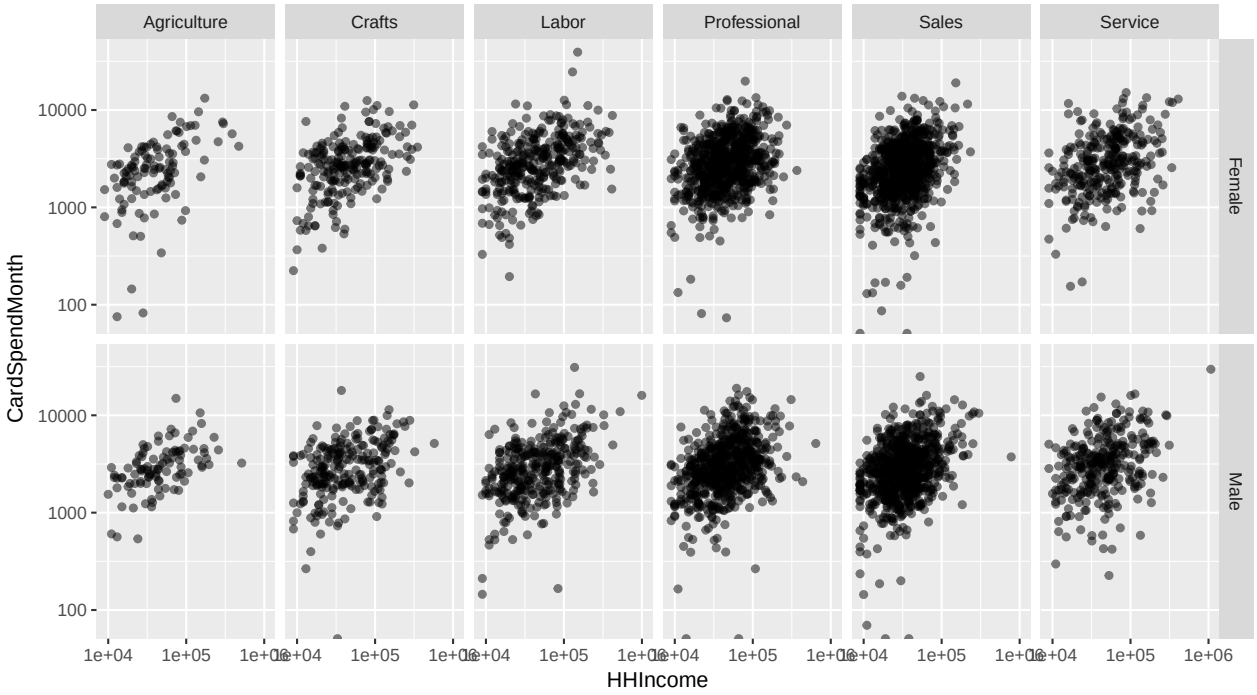
Facets are ways of grouping a data plot into multiple different pieces (subplots). This allows you to view a separate plot for each value in a categorical variable. You can construct a plot with multiple facets by using the `facet_wrap()` function. This will produce a "row" of subplots, one for each categorical variable (the number of rows can be specified with an additional argument):

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point(alpha = .5) +
  scale_x_log10() +
  scale_y_log10() +
  facet_wrap(~ JobCategory)
```



You can also `facet_grid` to facet your data by more than one categorical variable. Note that we use a tilde (~) in our `facet` functions. With `facet_grid` the variable to the left of the tilde will be represented in the rows and the variable to the right will be represented across the columns.

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth)) +
  geom_point(alpha = .5) +
  scale_x_log10() +
  scale_y_log10() +
  facet_grid(Gender ~ JobCategory)
```



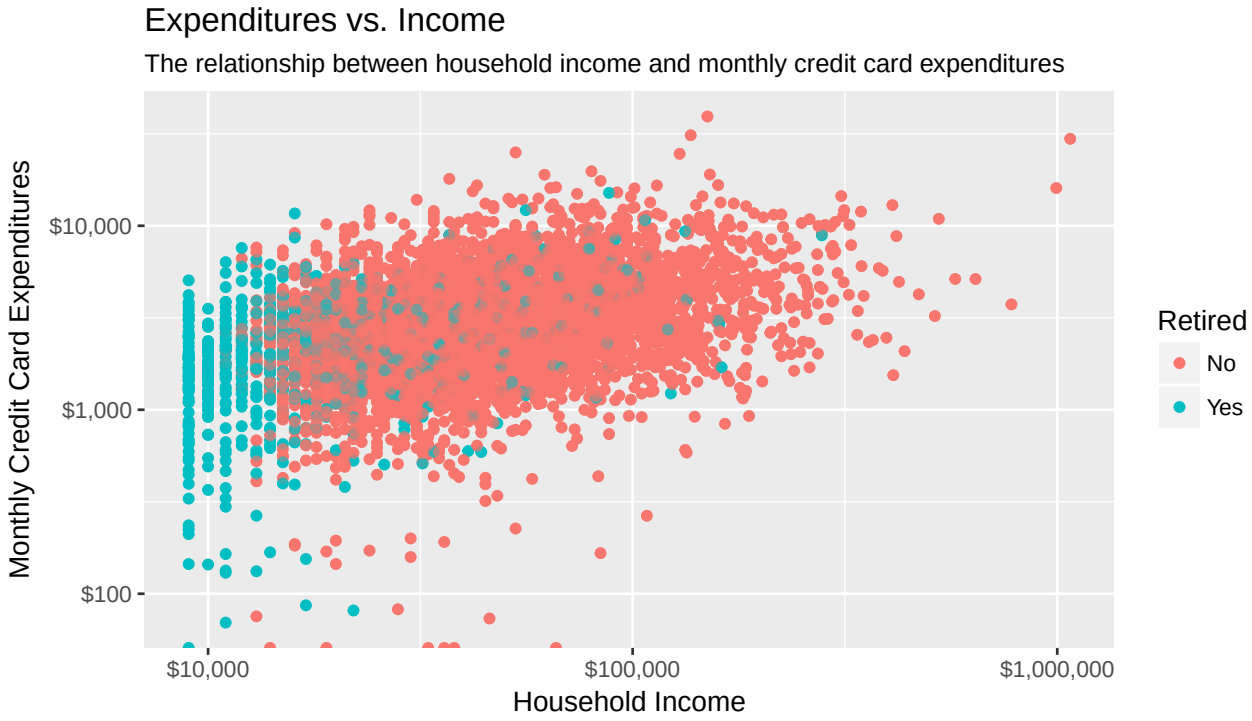
Labels & Annotations

Textual labels and annotations (on the plot, axes, geometry, and legend) are an important part of making a

plot understandable and communicating information. Although not an explicit part of the Grammar of Graphics (the would be considered a form of geometry), `ggplot` makes it easy to add such annotations.

You can add titles and axis labels to a chart using the `labs()` function (not `labels`, which is a different R function!):

```
ggplot(customer, aes(x = HHIncome, y = CardSpendMonth, color = Retired)) +
  geom_point() +
  geom_point(alpha = .5) +
  scale_x_log10(labels = scales::dollar) +
  scale_y_log10(labels = scales::dollar) +
  labs(title = "Expenditures vs. Income",
       subtitle = "The relationship between household income and monthly credit card expenditures",
       x = "Household Income",
       y = "Monthly Credit Card Expenditures")
```



Additional Resources on `ggplot2`

This gets you started with `ggplot2`; however, this a lot more to learn. Future tutorials illustrate how to convert many common forms of visualization (i.e. histograms, bar charts, line charts) and turn them into advanced, publication worthy graphics. Furthermore, the following resources provide additional avenues to learn more:

- gglot2 Documentation (particularly the function reference)
- ggplot2 Cheat Sheet (see also here)
- Data Visualization portion of R for Data Science Book
- A Layered Grammar of Graphics (Wickham)

Other Visualization Libraries

`ggplot2` is easily the most popular library for producing data visualizations in R. That said, ggplot2 is used to produce **static** visualizations: unchanging "pictures" of plots. Static plots are great for for **explanatory visualizations**: visualizations that are used to communicate some information—or more commonly, an argument about that information. All of the above visualizations have been ways for us to explain and demonstrate an argument about the data (e.g., the relationship between car engines and fuel efficiency).

Data visualizations can also be highly effective for **exploratory analysis**, in which the visualization is used as a way to ask and answer questions about the data (rather than to convey an answer or argument). While it is perfectly feasible to do such exploration on a static visualization, many explorations can be better served with **interactive visualizations** in which the user can select and change the view and presentation of that data in order to understand it.

While `ggplot2` does not directly support interactive visualizations, there are a number of additional R libraries that provide this functionality, including:

- `ggvis` is a library that uses the Grammar of Graphics (similar to ggplot), but for interactive visualizations.
- `plotly` is a open-source library for developing interactive visualizations. It provides a number of "standard" interactions (pop-up labels, drag to pan, select to zoom, etc) automatically. Moreover, it is possible to take a `ggplot2` plot and wrap it in Plotly in order to make it interactive. Plotly has many examples to learn from, though a less effective set of documentation.
- `htmlwidgets` provides a way to utilize a number of JavaScript interactive visualization libraries. JavaScript is the programming language used to create interactive websites (HTML files), and so is highly specialized for creating interactive experiences.

Exercises

Use visualizations to answer the following questions:

1. Does the distribution of *EducationYears* appear normally distributed?
2. Is distribution of *EducationYears* across *JobCategory* fairly consistent?
3. Are there more "younger" customers than "older" customers?
4. Does there appear to be a relationship between *Age* and *CardSpendMonth*?
5. Do certain credit card holders appear to spend more per month than others? Does this differ across gender or region?

Basic Statistical Inference

Hypothesis testing is a useful statistical tool that can be used to draw a conclusion about the population from a sample. Say for instance that you are interested in knowing if the average monthly card expenditures for your customers differs significantly from the national average within a well defined confidence level. Or maybe you want to understand if the average monthly card expenditures across regions or job categories differ. These questions can be answered with fundamental hypothesis testing.

Comparing Two Groups with a t-test

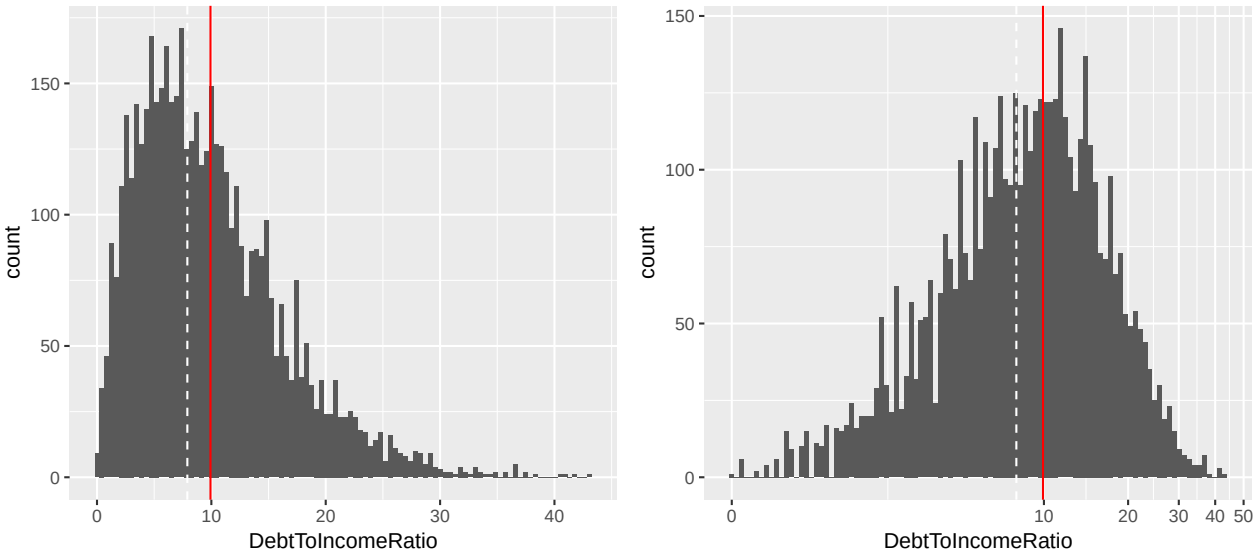
One of the most common tests in statistics, the t-test, is used to determine whether the means of two groups are equal to each other. The assumption for the test is that both groups are sampled from normal distributions with equal variances. The null hypothesis is that the two means are equal, and the alternative is that they are not. There is also a widely used modification of the t-test, known as Welch's t-test that adjusts the number of degrees of freedom when the variances are thought not to be equal to each other. This section covers the basics of performing t-tests in R.

One-sample t-test

The one-sample t-test compares a sample's mean with a known value, when the variance of the population is unknown. Consider we want to assess our customer's debt-to-income ratio and compare it to a certain value. For example, let's assume the nation-wide average of debt-to-income ratio is 7.9 and we want to know if our customer base is significantly different than the national average; in particular we want to test if our average debt-to-income ratio is greater than the national average because if it is than that means we are potentially taking on more risk than necessary.

The average debt-to-income ratio for our customers is 9.9 (red) and when we compare this to the national average (white) it appears significantly higher even when we adjust for non-normality in our distribution. However, we need to test to determine if this is statistically significant or within our mean confidence interval.

```
p1 <- ggplot(customer, aes(DebtToIncomeRatio)) +
  geom_histogram(bins = 100) +
  geom_vline(xintercept = mean(customer$DebtToIncomeRatio), color = "red") +
  geom_vline(xintercept = 7.9, color = "white", linetype = "dashed")

p2 <- ggplot(customer, aes(DebtToIncomeRatio)) +
  geom_histogram(bins = 100) +
  geom_vline(xintercept = mean(customer$DebtToIncomeRatio), color = "red") +
  geom_vline(xintercept = 7.9, color = "white", linetype = "dashed") +
  scale_x_continuous(trans = "log1p")

gridExtra::grid.arrange(p1, p2, ncol = 2)
```



Although the Central Limit Theorem provides some robustness to the normality assumption, it is important to know that we can test our data a couple different ways to provide a comprehensive conclusion. First, we use `t.test` to test if our average is *greater than* the national average of 7.9. Based on our results this suggests that yes, our customers' debt-to-income is greater than the national average.

```
t.test(customer$DebtToIncomeRatio, mu = 7.9, alternative = "greater")
##
##  One Sample t-test
##
## data:  customer$DebtToIncomeRatio
## t = 22.191, df = 4892, p-value < 2.2e-16
## alternative hypothesis: true mean is greater than 7.9
```

```
## 95 percent confidence interval:
##  9.774128      Inf
## sample estimates:
## mean of x
##  9.924198
```

However, due to the non-normality concerns we need to perform this test in two additional ways to ensure our results are not being biased due to assumption violations. We can perform the test with `t.test` and transform[13] our data and we can also perform the non-parametric test with the `wilcox.test` function. Both results, provided below, have p-value $< .05$ which supports our initial results suggesting that our clients tend to be more leveraged than the national average!

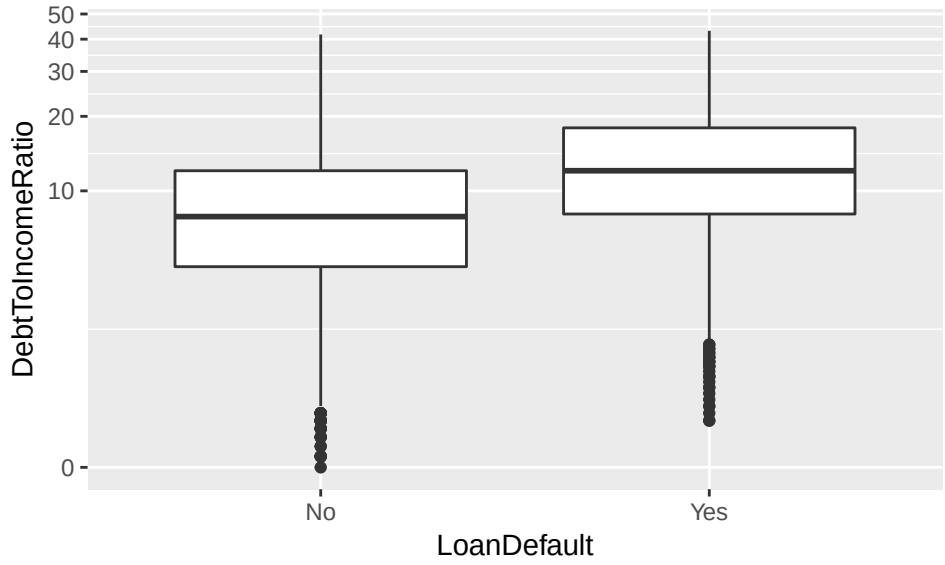
```
t.test(log1p(customer$DebtToIncomeRatio), mu = log1p(7.9), alternative = "greater")
##
##  One Sample t-test
##
## data:  log1p(customer$DebtToIncomeRatio)
## t = 2.6954, df = 4892, p-value = 0.003528
## alternative hypothesis: true mean is greater than 2.186051
## 95 percent confidence interval:
##  2.195547      Inf
## sample estimates:
## mean of x
##  2.210422

wilcox.test(customer$DebtToIncomeRatio, mu = 7.9, alternative = "greater")
##
##  Wilcoxon signed rank test with continuity correction
##
## data:  customer$DebtToIncomeRatio
## V = 7488400, p-value < 2.2e-16
## alternative hypothesis: true location is greater than 7.9
```

Two-Sample t-test

Now let's say we want to compare the differences between the debt-to-income ratio between those customers who have defaulted and those who have not. Visual assessment supports our hunch that defaulters are more leveraged than non-defaulters; however, we need to confirm this statistically.

```
ggplot(customer, aes(LoanDefault, DebtToIncomeRatio)) +
  geom_boxplot() +
  scale_y_continuous(trans = "log1p")
```

[13] Here we use `log1p` because we have many customers that have debt-to-income equal to 0. `log(0)`, `log10(0)`, and `log2(0)` all generate `-Inf` values whereas `log1p(0)` will return 0.



Here, we want to perform a two-sample t-test to see if they are statistically different. First, I test with a normal `t.test` without any distribution transformations. Note the slightly different syntax. Here, `DebtToIncomeRatio ~ LoanDefault` is saying to compare the debt-to-income ratio by the categories in `LoanDefault` ("Yes" or "No"). We see that this `t.test` suggests that our defaulters have a higher debt-to-income ratio that is statistically significant.

```
t.test(DebtToIncomeRatio ~ LoanDefault, data = customer)
##
##  Welch Two Sample t-test
##
## data:  DebtToIncomeRatio by LoanDefault
## t = -18.957, df = 1563.3, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -4.964772 -4.033700
## sample estimates:
##  mean in group No mean in group Yes
##          8.875939         13.375175
```

We can also test this with our data transformed and with the `wilcox.test`. Both support our initial findings that the the debt-to-income for the loan defaults is greater than that for non-defaulters. . . not a big surprise!

```
t.test(log1p(DebtToIncomeRatio) ~ LoanDefault, data = customer)
##
##  Welch Two Sample t-test
##
## data:  log1p(DebtToIncomeRatio) by LoanDefault
## t = -19.645, df = 1951.2, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.4357921 -0.3566785
## sample estimates:
##  mean in group No mean in group Yes
##          2.118105          2.514340

wilcox.test(DebtToIncomeRatio ~ LoanDefault, data = customer)
##
```

```
##  Wilcoxon rank sum test with continuity correction
##
## data:  DebtToIncomeRatio by LoanDefault
## W = 1336900, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
```

Suppose we wish to extend this test. Rather than just test for differences between two groups, what if we wish to test whether the mean value is the same across multiple categories. For example, what if we desire to test if the mean diamond price was the same across all cut categories. Here we turn to analysis of variance (ANOVA).

Comparing Multiple Groups with ANOVA

Let's assume we want to test if the monthly expenditures differ by credit card type (i.e. "Master Card", "Visa", "American Express", etc.)

ANOVA will test the hypotheses:

- $H_0 : \mu_{Mast} = \mu_{Visa} = \mu_{Disc} = \mu_{Othe} = \mu_{AMEX}$
- H_a : Not all the population means are equal

To perform ANOVA in R we can use the `aov` function. The summary of our `anova` object shows us several statistics. The primary one of interest is the `Pr(>F)`, which is our p-value. Here our p-value $<$ 2e16, which is extremely small and suggests that not all credit card types have equal average monthly expenditures. It basically says that at least one of the credit card mean monthly expenditures differs from another credit card; however, this doesn't tell us which one.

```
anova <- aov(CardSpendMonth ~ CreditCard, data = customer)
summary(anova)
##               Df    Sum Sq   Mean Sq F value Pr(>F)
## CreditCard     4 3.279e+09 819627015   151.5 <2e-16 ***
## Residuals   4888 2.645e+10   5410404
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

To identify which credit card differs with regard to their mean monthly expenditures we can use the `TukeyHSD` function. The output of `TukeyHSD` provides the pairwise differences (`diff`) for the cut categories, the 95% confidence range (`lwr`, `upr`) and the p-value (`p adj`) for that difference.

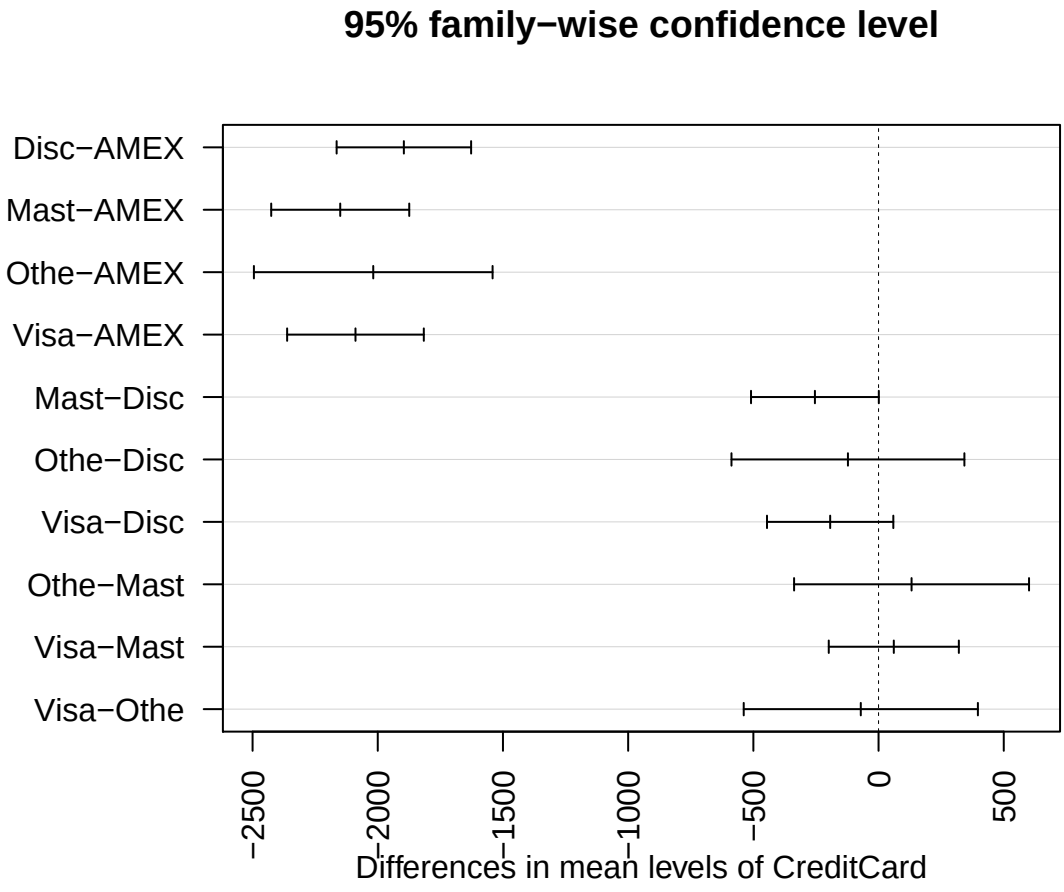
Below we see that average monthly expenditures for American Express (AMEX) is statistically different than all the other credit cards. Furthermore, we see that there is no statistically significant difference between the remaining credit cards (although the difference between Mastercard and Discover is borderline). This suggests that AMEX customers, on average, have higher monthly expenditures than all other customers. This is not a surprise since AMEX customers tend to be businesses.

```
TukeyHSD(anova)
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = CardSpendMonth ~ CreditCard, data = customer)
##
## $CreditCard
##                  diff        lwr          upr     p adj
## Disc-AMEX -1895.73084 -2164.2969 -1627.164766 0.0000000
## Mast-AMEX -2149.87228 -2425.4699 -1874.274609 0.0000000
## Othe-AMEX -2018.06055 -2494.6608 -1541.460311 0.0000000
```

```
## Visa-AMEX -2089.01312 -2361.9350 -1816.091245 0.0000000
## Mast-Disc  -254.14144  -509.2889     1.005982 0.0514940
## Othe-Disc  -122.32972  -587.4038   342.744388 0.9524918
## Visa-Disc  -193.28229  -445.5371    58.972512 0.2240931
## Othe-Mast   131.81172  -337.3580   600.981483 0.9401445
## Visa-Mast    60.85915  -198.8692   320.587520 0.9686100
## Visa-Othe   -70.95257  -538.5555   396.650406 0.9938475
```

We can easily view these differences and their confidence intervals with `plot`. This is plotting the pair-wise comparisons in the same order as the table output above.

```
diff <- TukeyHSD(anova)

# adjust margins in plot
par(oma=c(0,2,0,0))

# plot results
plot(diff, las = 2)
```



Exercises

1. Is the average age of our customers differ from 50 (p-value < 0.05)?
2. Does the average age of our *female* customers differ from the *male* customers?
3. Does the average age of our *retired* customers differ from the *non-retired* customers?
4. Do customers with *active lifestyles* tend to have longer or shorter *card tenure*?

5. Do any credit card (Mastercard, Discover, AMEX, etc.) have longer *card tenure* than other credit cards?

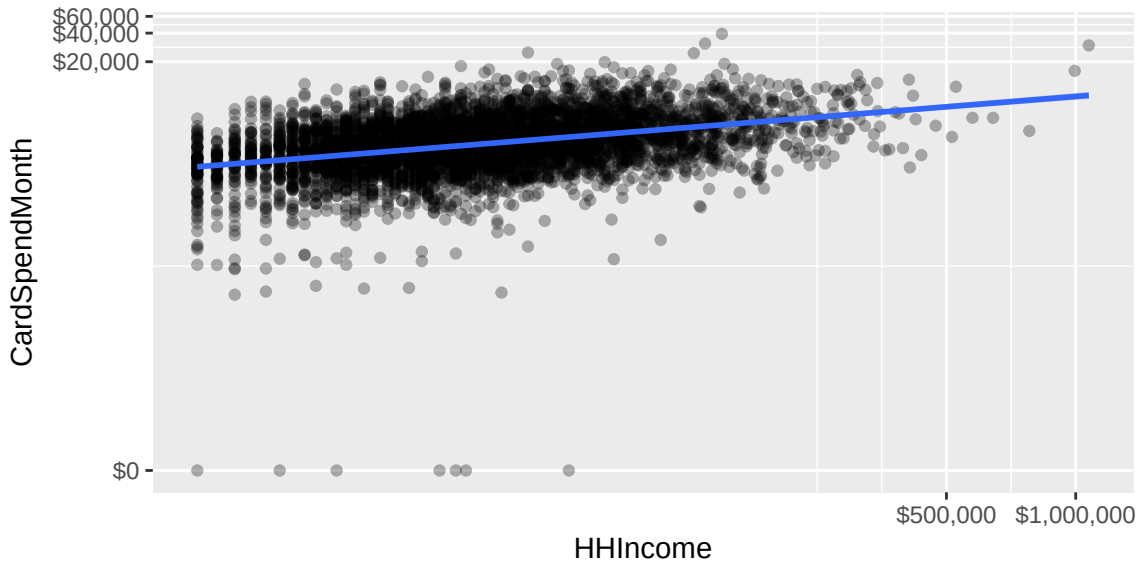
Identifying Relationships

A big part of EDA is to try and understand relationships between variables. Two of the most common ways to identify relationships is through *correlation* analysis and *Linear regression*. This section will illustrate how to employ both analytic methods in R.

Correlation

We saw earlier that there appears to be a relationship between household income (*HHIncome*) and monthly credit card expenditures (*CardSpendMonth*). When we transform these variables logarithmically this relationship becomes a little more apparent:

```
ggplot(customer, aes(HHIncome, CardSpendMonth)) +
  geom_point(alpha = .3) +
  scale_x_continuous(trans = "log1p", labels = scales::dollar) +
  scale_y_continuous(trans = "log1p", labels = scales::dollar) +
  geom_smooth(method = "lm")
```



R provides multiple functions to analyze correlations. To calculate the correlation between two variables we use `cor()`. Note that there is an argument `method =` which allows us to specify if we want to use "pearson", "kendall", or "spearman". Pearson is the default method so we do not need to specify for that option.

```
# Filter NAs to get correlation for complete observations
cor(customer$HHIncome, customer$CardSpendMonth)
## [1] 0.3481489

cor(log1p(customer$HHIncome), log1p(customer$CardSpendMonth))
## [1] 0.3668594
```

Unfortunately `cor()` only provides the r coefficient(s) and does not test for significance nor provide confidence intervals. To get these parameters for a simple two variable analysis you can use `cor.test()`. In our example we see that the p-value is significant and the 95% confidence interval confirms this as the range

does not contain zero. This suggests the correlation between household income and monthly card expenditures is $r = 0.37$ with 95% confidence of being between 0.34 and 0.39.

```
cor.test(log1p(customer$HHIncome), log1p(customer$CardSpendMonth))
##
##  Pearson's product-moment correlation
##
## data:  log1p(customer$HHIncome) and log1p(customer$CardSpendMonth)
## t = 27.58, df = 4891, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.3423580 0.3908622
## sample estimates:
##       cor
## 0.3668594
```

You can also incorporate this into piped functions to streamline your analyses. For instance, you can calculate the correlation (and p-value) within `summarize`.

```
customer %>%
  summarize(
    r = cor(log1p(HHIncome), log1p(CardSpendMonth)),
    p_value = cor.test(log1p(HHIncome), log1p(CardSpendMonth))$p.value)
## # A tibble: 1 x 2
##           r       p_value
##       <dbl>         <dbl>
## 1 0.3668594 9.406881e-156
```

Why is this important? Because you may want to understand relationships at different categorical levels. For instance, does the correlation between *HHIncome* and *CardSpendMonth* differ depending on the type of card used? To answer this we just incorporate a `group_by` statement to calculate the correlations and associated p-values for each cut. We see some slight differences in the strength of correlation based on credit card type.

```
customer %>%
  group_by(CreditCard) %>%
  summarize(
    r = cor(log1p(HHIncome), log1p(CardSpendMonth)),
    p_value = cor.test(log1p(HHIncome), log1p(CardSpendMonth))$p.value)
## # A tibble: 5 x 3
##   CreditCard         r      p_value
##        <chr>     <dbl>        <dbl>
## 1       AMEX 0.4456491 1.486165e-48
## 2       Disc 0.3807071 1.308195e-46
## 3       Mast 0.3330449 1.122489e-31
## 4       Othe 0.4115825 2.792979e-10
## 5       Visa 0.3278580 5.478730e-32
```

Linear Regression

Although correlation provides a measurement of the relationship strength between *HHIncome* and *CardSpendMonth*, it does not provide us with much more information. To gain more insights we can use linear regression.

Preparing the data

Initial discovery of relationships is usually done with a training set while a test set is used for evaluating whether the discovered relationships hold. More formally, a training set is a set of data used to discover potentially predictive relationships. A test set is a set of data used to assess the strength and utility of a predictive relationship. In a later tutorial we will cover more sophisticated ways for training, validating, and testing predictive models but for the time being we'll use a conventional 60% / 40% split where we training our model on 60% of the data and then test the model performance on 40% of the data that is withheld.

```
set.seed(123)
sample <- sample(c(TRUE, FALSE), nrow(customer), replace = T, prob = c(0.6, 0.4))
train <- customer[sample, ]
test <- customer[!sample, ]
```

Fitting the model

To perform a simple linear regression model that models *CardSpendMonth* as a function of *HHIncome*...

$$log1p(CardSpendMonth) = \beta_0 + \beta_1 \times log1p(HHIncome) + \epsilon$$

we use the `lm()` function to perform the regression and then `summary()` to view the results. We see that our coefficient estimate for *HHIncome* is statistically significant; however, we also see that our model has a low adjusted R^2. We can also use `confint` to get confidence intervals on our coefficients.

```
model1 <- lm(log1p(CardSpendMonth) ~ log1p(HHIncome), data = train)

# get a summary of our model
summary(model1)
##
## Call:
## lm(formula = log1p(CardSpendMonth) ~ log1p(HHIncome), data = train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.8802 -0.3583  0.0365  0.4170  2.2186
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)      4.19942    0.18043   23.27   <2e-16 ***
## log1p(HHIncome)  0.34904    0.01696   20.58   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6862 on 2943 degrees of freedom
## Multiple R-squared:  0.1258, Adjusted R-squared:  0.1255
## F-statistic: 423.4 on 1 and 2943 DF,  p-value: < 2.2e-16

# get confidence intervals on our coefficients
confint(model1)
##                     2.5 %    97.5 %
## (Intercept)     3.8456445 4.5532020
## log1p(HHIncome) 0.3157835 0.3823048
```

Earlier we saw that the type of credit card had an impact on the average monthly expenditures. We can add the *CreditCard* variable into our model to see if that increases our models performance.

```
model2 <- lm(log1p(CardSpendMonth) ~ log1p(HHIncome) + CreditCard, data = train)
summary(model2)
##
## Call:
## lm(formula = log1p(CardSpendMonth) ~ log1p(HHIncome) + CreditCard,
##     data = train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.8000 -0.3315  0.0377  0.3954  1.9433
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)      4.68034    0.17908  26.135  < 2e-16 ***
## log1p(HHIncome)  0.34178    0.01655  20.653  < 2e-16 ***
## CreditCardDisc  -0.53463    0.03614 -14.793  < 2e-16 ***
## CreditCardMast  -0.51245    0.03679 -13.929  < 2e-16 ***
## CreditCardOthe  -0.35780    0.06272  -5.704 1.28e-08 ***
## CreditCardVisa  -0.48453    0.03665 -13.219  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.6561 on 2939 degrees of freedom
## Multiple R-squared:  0.2019, Adjusted R-squared:  0.2005
## F-statistic: 148.7 on 5 and 2939 DF,  p-value: < 2.2e-16
```

We may also want to see if an interaction effect impacts the relationship. We can do that by using * rather than + on the right-hand side of the tilde (~).

```
model3 <- lm(log1p(CardSpendMonth) ~ log1p(HHIncome) * CreditCard, data = train)
summary(model3)
##
## Call:
## lm(formula = log1p(CardSpendMonth) ~ log1p(HHIncome) * CreditCard,
##     data = train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.7985 -0.3296  0.0370  0.3903  1.9327
##
## Coefficients:
##                                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)                      5.11100    0.37858  13.500   <2e-16 ***
## log1p(HHIncome)                  0.30151    0.03531   8.540   <2e-16 ***
## CreditCardDisc                  -1.08409    0.50365  -2.152   0.0314 *
## CreditCardMast                  -1.21057    0.52180  -2.320   0.0204 *
## CreditCardOthe                  -2.37028    1.08092  -2.193   0.0284 *
## CreditCardVisa                  -0.69407    0.52719  -1.317   0.1881
## log1p(HHIncome):CreditCardDisc   0.05127    0.04677   1.096   0.2731
## log1p(HHIncome):CreditCardMast   0.06565    0.04901   1.339   0.1805
## log1p(HHIncome):CreditCardOthe   0.19397    0.10438   1.858   0.0632 .
## log1p(HHIncome):CreditCardVisa   0.01915    0.04969   0.385   0.6999
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.656 on 2935 degrees of freedom
## Multiple R-squared:  0.2032, Adjusted R-squared:  0.2008
## F-statistic: 83.17 on 9 and 2935 DF,  p-value: < 2.2e-16
```

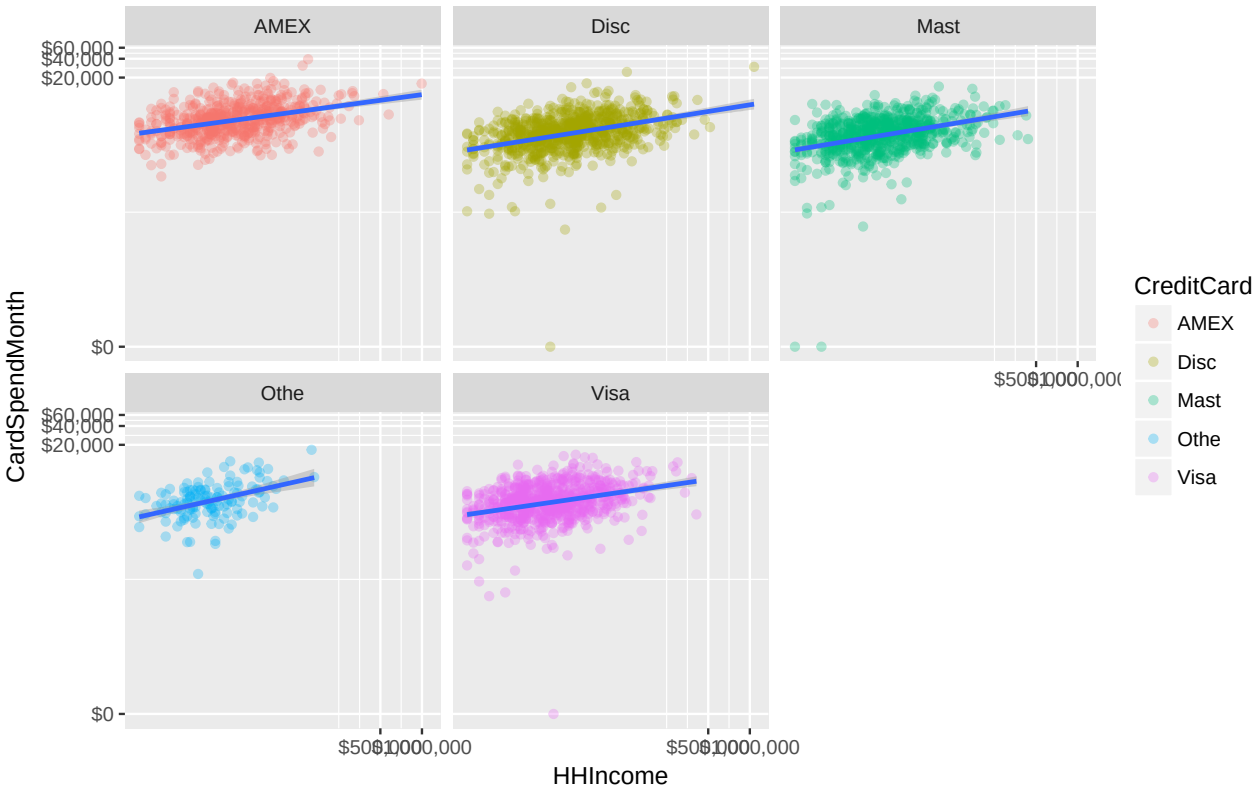
Model diagnostics

To compare if the performance is improving as we add variables we should not just rely on R^2 values. We can use `anova`, which performs the Chi-square test to compare two models (i.e. it tests whether reduction in the residual sum of squares are statistically significant or not). Here we see that `model2` is an improvement over `model1` but `model3` does not improve performance over `model2`.

```
# comparing model1 and model2
anova(model1, model2, test = "Chisq")
## Analysis of Variance Table
##
## Model 1: log1p(CardSpendMonth) ~ log1p(HHIncome)
## Model 2: log1p(CardSpendMonth) ~ log1p(HHIncome) + CreditCard
##   Res.Df    RSS Df Sum of Sq  Pr(>Chi)
## 1   2943 1385.9
## 2   2939 1265.2  4    120.68 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

# comparing model2 and model3
anova(model2, model3, test = "Chisq")
## Analysis of Variance Table
##
## Model 1: log1p(CardSpendMonth) ~ log1p(HHIncome) + CreditCard
## Model 2: log1p(CardSpendMonth) ~ log1p(HHIncome) * CreditCard
##   Res.Df    RSS Df Sum of Sq Pr(>Chi)
## 1   2939 1265.2
## 2   2935 1263.1  4    2.0657   0.3084
```

We can visualize `model2` within our data using `ggplot2`:

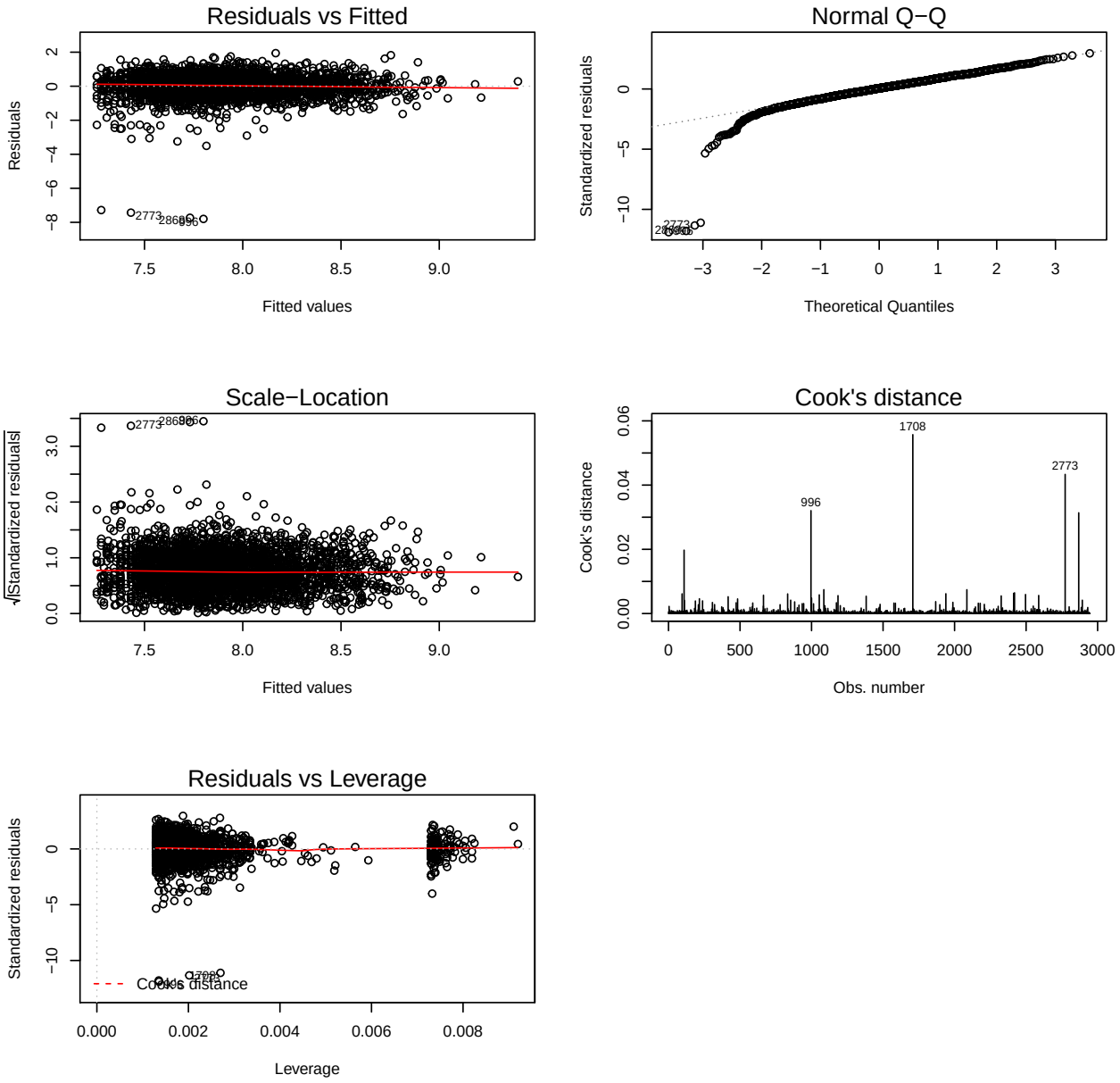
```
ggplot(train, aes(HHIncome, CardSpendMonth)) +
  geom_point(aes(color = CreditCard), alpha = .3) +
  geom_smooth(method = "lm") +
  scale_x_continuous(trans = "log1p", labels = scales::dollar) +
  scale_y_continuous(trans = "log1p", labels = scales::dollar) +
  facet_wrap(~ CreditCard)
```



Ordinary least squares regression relies on several assumptions, including that the residuals are normally distributed and homoskedastic, the errors are independent and the relationships are linear. Investigating residuals helps us to assess these assumptions for our model:

`plot`ting our model allows us to look at five different residual plots. Adjusting the `which` argument will allow you to plot each one individually. This illustrates some interesting points.

```
par(mfrow = c(3, 2)) #optional
plot(model2, which = c(1:5))
```



Looking at the Q-Q plot we see that our residuals are approximately normal with the exception of those on the far left. We also see in the Cook's distance plot several observations that have very high distances (these are influential outlier observations that deserve closer attention). We can look into these observations more closely with the `augment` function, which extracts several model diagnostics. Here we filter for those observations that have the the highest Cook's distance and we see that many of these observations have low monthly card expenditures ($0 - $500) but moderate income. If you filter for the lowest standardized residuals you would find the same results. Thus, our model is being influenced some by customers with very low or no credit card expenditures.

```
library(broom)

augment(model2) %>%
  top_n(10, wt = .cooksd)
##    log1p.CardSpendMonth. log1p.HHIncome. CreditCard  .fitted    .se.fit
## 1               5.150977       10.085851       Othe 7.769700 0.05615841
## 2               0.000000       10.545368       Visa 7.800025 0.02416621
```

```
## 3               6.257668       10.373522       Othe 7.868021 0.05607182
## 4               0.000000        9.105091       Mast 7.279842 0.03409398
## 5               4.309456       10.736418       Disc 7.815222 0.02359401
## 6               4.334673        9.472782       Visa 7.433434 0.02925393
## 7               4.983607        9.105091       Disc 7.257664 0.03667549
## 8               6.368016       10.373522       Othe 7.868021 0.05607182
## 9               0.000000        9.546884       Mast 7.430839 0.02947784
## 10              0.000000       10.491302       Disc 7.731446 0.02412640
##       .resid        .hat    .sigma     .cooksd .std.resid
## 1  -2.618723 0.007326139 0.6544284 0.019739476  -4.005983
## 2  -7.800025 0.001356634 0.6402278 0.032042634 -11.896346
## 3  -1.610354 0.007303565 0.6555446 0.007441129  -2.463405
## 4  -7.279842 0.002700231 0.6422935 0.055704186 -11.110457
## 5  -3.505766 0.001293150 0.6530232 0.006169262  -5.346712
## 6  -3.098761 0.001987990 0.6537225 0.007420160  -4.727625
## 7  -2.274058 0.003124623 0.6548757 0.006295252  -3.471393
## 8  -1.500005 0.007303565 0.6556344 0.006456275  -2.294602
## 9  -7.430839 0.002018538 0.6417131 0.043327347 -11.337034
## 10 -7.731446 0.001352168 0.6405114 0.031377737 -11.791725
```

Making Predictions

To make predictions with our data on a new (or test) data set we use `add_predictions` from the `modelr` package. After I add the predictions here I use `mutate` and `expm1` to transform our prediction values back so we can compare them to the *CardSpendMonth* values.

```
library(modelr)

(test <- test %>%
    select(CardSpendMonth, HHIncome, CreditCard) %>%
    add_predictions(model2) %>%
    mutate(pred_trans = expm1(pred)))
## # A tibble: 1,948 x 5
##    CardSpendMonth HHIncome CreditCard     pred pred_trans
##             <dbl>    <dbl>      <chr>    <dbl>      <dbl>
##  1          426.0    15000       Visa 7.482340   1775.393
##  2         3409.9    20000       Visa 7.580659   1958.920
##  3         2551.0    23000       Disc 7.578325   1954.350
##  4         5927.0    97000       AMEX 8.604837   5456.994
##  5         4889.7    47000       Disc 7.822572   2495.319
##  6         5343.6    73000       Disc 7.973060   2900.724
##  7         2974.7    23000       Mast 7.600503   1998.201
##  8           81.1    22000       Visa 7.613233   2023.813
##  9         2719.8    35000       AMEX 8.256443   3851.367
## 10         2677.1    28000       AMEX 8.180179   3568.494
## # ... with 1,938 more rows
```

The primary concern is to assess if the out-of-sample mean squared error (MSE), also known as the mean squared prediction error, is substantially higher than the in-sample mean square error, as this is a sign of deficiency in the model. The MSE is computed as

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

We can easily compare the test sample MSE to the training sample MSE with the following. We see that the test MSE is actually lower than the training MSE so we can be confident that we are not overfitting our model.

```
# test MSE
test %>%
  summarise(MSE = mean((CardSpendMonth - pred_trans)^2))
## # A tibble: 1 x 1
##       MSE
##     <dbl>
## 1 4504973

# training MSE
train %>%
  select(CardSpendMonth, HHIncome, CreditCard) %>%
  add_predictions(model2) %>%
  mutate(pred_trans = expm1(pred)) %>%
  summarise(MSE = mean((CardSpendMonth - pred_trans)^2))
## # A tibble: 1 x 1
##       MSE
##     <dbl>
## 1 5125697
```

However, this practice becomes more powerful when you are comparing multiple models. For example, we could easily compare the out-of-sample MSE for `model1`, `model2`, and `model3`. Note that we use `gather_predictions` to gather predictions from several models. We can then incorporate a `group_by` statement to compute the MSE for each model and we see that `model2` minimizes the out-of-sample error.

```
test %>%
  select(CardSpendMonth, HHIncome, CreditCard) %>%
  gather_predictions(model1, model2, model3) %>%
  mutate(pred_trans = expm1(pred)) %>%
  group_by(model) %>%
  summarise(MSE = mean((CardSpendMonth - pred_trans)^2))
## # A tibble: 3 x 2
##    model     MSE
##    <chr>   <dbl>
## 1 model1 5273049
## 2 model2 4504973
## 3 model3 4543911
```

Exercises

1. What is the correlation between *household income* and *years of education* for our customers?
2. Is there a strong relationship between *employment length* and *household income*?
3. Do either of these correlations differ based on *region* or *gender*?
4. Filter out customers that have $0 *monthly card expenditures*. Refit `model2` and see if this improves model performance and residual diagnostics.
5. Build onto `model2` by adding variable such as *gender*, *region*, and *CardTenure*. Do these variables improve upon `model2`? Do they improve residual diagnostics? What is the out-of-sample MSE for these more complex models?

Additional Resources

This will get you up and running with exploratory data analysis. Keep in mind that there is a *lot* more you can dig into so the following resources will help you learn more:

- R for Data Science
- Practical Data Science with R
- Graphical Data Analysis with R
- Discoverying Statistics Using R
- An Introduction to Statistical Learning
- Applied Predictive Modeling

Case Study: Further Exploration of Customer Data

This module provides an opportunity to put into practice many of the techniques discussed in the previous modules. The goal is to think through the tool(s) you'll need and use them properly to answer the questions. This case study continues to use the *CustomerData_Merrimack.xlsx* data and, although not comprehensive, provides you the opportunity to perform the entire analytic process to include:

1. Organizing your files
2. Importing the data
3. Dealing with missing values
4. Understanding numerical and categorical variables through descriptive statistics
5. Using visualization to understand your data
6. Perform basic statistical inference on variables
7. Identifying relationships between variables

Organizing Your Files

1. Create a folder on your computer titled *Customer Data Case Study*. Within R, set your working directory to this folder.
2. Create a new .R file and save this as *case-study.R* (note how this now appears in your *Customer Data Case Study* folder).
3. Set up this .R script so that you are loading your packages at the top, and then create a section for each of the steps you perform in the following sections and annotate them thoroughly.

Importing the Data

1. Import the *CustomerData_Merrimack.xlsx* data.
2. What are the dimensions of this data set?
3. What are the names of all the variables?
4. What *type* of variables are in this data (i.e. character, integer, double)?

Dealing with Missing Values

1. How many missing values are in the data?
2. Which variables contain these missing values?
3. Omit all rows with missing values in them? Now how many observations does your data have?

Understanting Your Data Through Descriptive Statistics

Answer the following with numerical descriptive statistics:

1. Which *JobCategory* has the largest average *HHIncome*? Which has the lowest?
2. Is there a large difference in the number of earners in each *JobCategory*?
3. How many earners are in each *JobCategory* across the regions?
4. Does the average *HHIncome* vary across *JobCategory* and *Region*?
5. Do these customers tend to be married, single, or an equal mix?
6. Are married customers more educated than single customers?
7. Which age group is most likely to lease an automobile?
8. Which age group tends to have the greatest *CarValue*?
9. Which credit card has the highest default rate?

Understanting Your Data Through Visualization

Answer the following using base R or `ggplot2` visualizations:

1. Which *JobCategory* has the largest average *HHIncome*? Which has the lowest?
2. Is there a large difference in the number of earners in each *JobCategory*?
3. How many earners are in each *JobCategory* across the regions?
4. Does the average *HHIncome* vary across *JobCategory* and *Region*?
5. Do these customers tend to be married, single, or an equal mix?
6. Are married customers more educated than single customers?
7. Which age group is most likely to lease an automobile?
8. Which age group tends to have the greatest *CarValue*?
9. Which credit card has the highest default rate?

Perform Basic Statistics Inference

Using a p-value < 0.05 to signify statistical significance:

1. Does the average *HHIncome* differ by *JobCategory*?
2. Does the average *HHIncome* for the sales *JobCategory* differ by *Region*?
3. Are married customers more educated than non-married customers?
4. Does the average *CarValue* differ across the following categories (young < 30, middle 30-59, senior $>$ 60)?
5. Do customers with an *ActiveLifestyle* tend to have higher incomes than inactive customers? Do they tend to have larger *DebtToIncomeRatio*?

Identifying Relationships Between Variables

1. Is there a strong relationship between *CardItemsMonthly* and *CardSpendMonth* for our customers?
2. Is there a strong relationship between *PhoneCoTenure* and *VoiceLastMonth* for our customers?
3. Do either of these relationships differ based on region, gender, or credit card type? Do you find this interesting?
4. Build a simple bivariate regression model to model *CardSpendMonth* as a function of *CardItemsMonthly*? Do the residuals suggest this model meets required assumptions? Does a log-log transformation of this model improve the residual fit? What is the out-of-sample MSE for this model?
5. Build a multivariate regression model with *CardSpendMonth* as a function of *HHIncome*, *CreditCard*, *gender*, *region*, and *CardTenure*. Does transforming (i.e. log, sqrt) any of these variables improve the residual diagnostics? Which variables are considered statistically significant? How would you interpret the coefficients for the statistically significant variables? What is the out-of-sample MSE for this model?

Case Study Results

The following illustrates *approaches* you could take to answer the questions. However, keep in mind there are several approaches you can take to get similar results.

Organizing Your Files

Q1. Create a folder on your computer titled *Customer Data Case Study*. Within R, set your working directory to this folder.
A1. Create a folder as you customarily do. Then set you working directory, within R, to this folder by selecting **Session » Set Working Directory » Choose Directory** and select the folder you just created.

Q2. Create a new .R file and save this as *case-study.R* (note how this now appears in your *Customer Data Case Study* folder).
A1. To create a new .R file go to **File » New File » R Script**

Q3. Set up this .R script so that you are loading your packages at the top, and then create a section for each of the steps you perform in the following sections and annotate them thoroughly.
A3. You can see a properly formatted .R script here.

Importing the Data

Q1. Import the *CustomerData_Merrimack.xlsx* data.
A2. Import the data with `readxl::read_xlsx`.

```
raw <- read_xlsx("data/CustomerData_Merrimack.xlsx")
```

Q2. What are the dimensions of this data set?
A2. This data has 5000 rows and 59 columns

```
dim(raw)
## [1] 5000   59
```

Q3. What are the names of all the variables?
A3. You can see the variable names using either `names()` or `str()`.

```
names(raw)
##  [1] "CustomerID"          "Region"              "TownSize"
##  [4] "Gender"              "Age"                 "EducationYears"
##  [7] "JobCategory"         "UnionMember"         "EmploymentLength"
## [10] "Retired"             "HHIncome"            "DebtToIncomeRatio"
## [13] "CreditDebt"          "OtherDebt"           "LoanDefault"
## [16] "MaritalStatus"       "HouseholdSize"       "NumberPets"
## [19] "NumberCats"          "NumberDogs"          "NumberBirds"
## [22] "HomeOwner"           "CarsOwned"           "CarOwnership"
## [25] "CarBrand"            "CarValue"            "CommuteTime"
## [28] "PoliticalPartyMem"   "Votes"               "CreditCard"
## [31] "CardTenure"          "CardItemsMonthly"    "CardSpendMonth"
## [34] "ActiveLifestyle"     "PhoneCoTenure"       "VoiceLastMonth"
## [37] "VoiceOverTenure"     "EquipmentRental"     "EquipmentLastMonth"
## [40] "EquipmentOverTenure" "CallingCard"         "WirelessData"
## [43] "DataLastMonth"       "DataOverTenure"      "Multiline"
## [46] "VM"                  "Pager"               "Internet"
## [49] "CallerID"            "CallWait"            "CallForward"
## [52] "ThreeWayCalling"     "EBilling"            "TVWatchingHours"
```

```
## [55] "OwnsPC"              "OwnsMobileDevice"    "OwnsGameSystem"
## [58] "OwnsFax"             "NewsSubscriber"
```

Q4. What *type* of variables are in this data (i.e. character, integer, double)?
A4. You can see the type of for each variable with `str()` or `sapply()`.

```
sapply(raw, typeof)
##          CustomerID              Region            TownSize
##         "character"            "double"         "character"
##              Gender                 Age      EducationYears
##         "character"            "double"            "double"
##         JobCategory         UnionMember    EmploymentLength
##         "character"         "character"            "double"
##             Retired            HHIncome   DebtToIncomeRatio
##         "character"            "double"            "double"
##          CreditDebt           OtherDebt         LoanDefault
##            "double"            "double"         "character"
##       MaritalStatus       HouseholdSize          NumberPets
##         "character"            "double"            "double"
##          NumberCats          NumberDogs         NumberBirds
##            "double"            "double"            "double"
##           HomeOwner           CarsOwned        CarOwnership
##            "double"            "double"         "character"
##            CarBrand            CarValue         CommuteTime
##         "character"            "double"         "character"
##   PoliticalPartyMem               Votes          CreditCard
##         "character"         "character"         "character"
##          CardTenure    CardItemsMonthly      CardSpendMonth
##            "double"            "double"            "double"
##     ActiveLifestyle       PhoneCoTenure      VoiceLastMonth
##         "character"            "double"            "double"
##     VoiceOverTenure     EquipmentRental  EquipmentLastMonth
##         "character"         "character"            "double"
## EquipmentOverTenure         CallingCard        WirelessData
##            "double"         "character"         "character"
##       DataLastMonth      DataOverTenure           Multiline
##            "double"            "double"         "character"
##                  VM               Pager            Internet
##         "character"         "character"         "character"
##            CallerID            CallWait         CallForward
##         "character"         "character"         "character"
##     ThreeWayCalling            EBilling     TVWatchingHours
##         "character"         "character"            "double"
##             OwnsPC    OwnsMobileDevice      OwnsGameSystem
##         "character"         "character"         "character"
##            OwnsFax      NewsSubscriber
##         "character"         "character"
```

Dealing with Missing Values

Q1. How many missing values are in the data?
A1. There are 124 missing values in this data.

```
sum(is.na(raw))
## [1] 124
```

Q2. Which variables contain these missing values?
A2. 8 of the 59 variables contain these missing values.

```
missing <- colSums(is.na(raw))
sort(missing[missing > 0], decreasing = TRUE)
##   NumberBirds        Gender   JobCategory     HomeOwner HouseholdSize
##            34            33            15            13             8
##    NumberDogs    NumberCats    NumberPets
##             8             7             6
```

Q3. Omit all rows with missing values in them? Now how many observations does your data have? **A3**. After removing missing values we now have 4,893 rows for our 59 variables.

```
customers <- na.omit(raw)
dim(customers)
## [1] 4893   59
```

Understanting Your Data Through Descriptive Statistics

Answer the following with numerical descriptive statistics:

Q1. Which *JobCategory* has the largest average *HHIncome*? Which has the lowest?
A1. We see that the Labor job category has the largest average *HHIncome* while Sales has the lowest.

```
customers %>%
  group_by(JobCategory) %>%
  summarize(avg_HHIncome = mean(HHIncome)) %>%
  arrange(desc(avg_HHIncome))
## # A tibble: 6 x 2
##    JobCategory avg_HHIncome
##          <chr>        <dbl>
## 1        Labor     67353.20
## 2      Service     65398.36
## 3       Crafts     62984.30
## 4  Agriculture     62280.95
## 5 Professional     57618.28
## 6        Sales     40450.91
```

Q2. Is there a large difference in the number of earners in each *JobCategory*?
A2. There is a pretty subtantial difference in the number of earners across the categories.

```
customers %>%
  count(JobCategory) %>%
  arrange(desc(n))
## # A tibble: 6 x 2
##    JobCategory     n
##          <chr> <int>
## 1        Sales  1599
## 2 Professional  1357
## 3        Labor   671
## 4      Service   610
## 5       Crafts   446
## 6  Agriculture   210
```

Q3. How many earners are in each *JobCategory* across the regions?
A3. We can inspect the number of earners in each category with the following:

```
customers %>%
  count(JobCategory, Region) %>%
  arrange(desc(n))
## Source: local data frame [30 x 3]
## Groups: JobCategory [6]
##
## # A tibble: 30 x 3
##     JobCategory Region     n
##           <chr>  <dbl> <int>
##  1        Sales      5   338
##  2        Sales      2   328
##  3        Sales      3   321
##  4        Sales      1   310
##  5        Sales      4   302
##  6 Professional      2   286
##  7 Professional      3   277
##  8 Professional      4   271
##  9 Professional      1   264
## 10 Professional      5   259
## # ... with 20 more rows
```

Q4. Does the average *HHIncome* vary across *JobCategory* and *Region*?
A4. Yes, there is a fair amount of variation in *HHIncome*. We can assess it a couple different ways:

```
customers %>%
  group_by(JobCategory, Region) %>%
  summarize(avg_HHIncome = mean(HHIncome)) %>%
  arrange(desc(avg_HHIncome))
## Source: local data frame [30 x 3]
## Groups: JobCategory [6]
##
## # A tibble: 30 x 3
##     JobCategory Region avg_HHIncome
##           <chr>  <dbl>        <dbl>
##  1        Labor      5     82582.09
##  2  Agriculture      3     81350.00
##  3      Service      4     77872.88
##  4        Labor      3     71279.41
##  5       Crafts      5     70769.23
##  6       Crafts      4     68371.13
##  7  Agriculture      5     65914.89
##  8 Professional      3     65750.90
##  9      Service      5     64496.30
## 10        Labor      4     64471.54
## # ... with 20 more rows

# or...
customers %>%
  group_by(JobCategory, Region) %>%
  summarize(avg_HHIncome = mean(HHIncome)) %>%
  group_by(JobCategory) %>%
  summarize(sd = sd(avg_HHIncome))
```

```
## # A tibble: 6 x 2
##    JobCategory        sd
##          <chr>     <dbl>
## 1  Agriculture 12290.077
## 2       Crafts  6800.815
## 3        Labor 10182.615
## 4 Professional  4641.507
## 5        Sales  2594.653
## 6      Service  7414.040
```

Q5. Do these customers tend to be married, single, or an equal mix?
A5. There are slightly more unmarried customers in this data.

```
customers %>%
  count(MaritalStatus) %>%
  mutate(pct = n / sum(n))
## # A tibble: 2 x 3
##   MaritalStatus     n       pct
##           <chr> <int>     <dbl>
## 1       Married  2348 0.4798692
## 2     Unmarried  2545 0.5201308
```

Q6. Are married customers more educated than single customers?
A6. Average education levels appears to be fairly consistent across marital status.

```
customers %>%
  group_by(MaritalStatus) %>%
  summarize(avg_Ed = mean(EducationYears))
## # A tibble: 2 x 2
##   MaritalStatus   avg_Ed
##           <chr>    <dbl>
## 1       Married 14.48637
## 2     Unmarried 14.61493
```

Q7. Which age group is most likely to lease an automobile?
A7. It appears that older customers have a higher percentage of leasing activity than owning or not having an automobile.

```
customers %>%
  group_by(Age) %>%
  count(CarOwnership) %>%
  mutate(pct_lease = round(n / sum(n), 2)) %>%
  filter(CarOwnership == "Lease") %>%
  arrange(desc(pct_lease))
## Source: local data frame [62 x 4]
## Groups: Age [62]
##
## # A tibble: 62 x 4
##      Age CarOwnership     n pct_lease
##    <dbl>        <chr> <int>     <dbl>
##  1    78        Lease    36      0.53
##  2    79        Lease    38      0.52
##  3    76        Lease    25      0.43
##  4    73        Lease    25      0.39
##  5    74        Lease    22      0.37
##  6    72        Lease    21      0.34
```

```
##  7    68        Lease    29      0.33
##  8    75        Lease    24      0.32
##  9    77        Lease    22      0.32
## 10    65        Lease    17      0.24
## # ... with 52 more rows
```

Q8. Which age group tends to have the greatest *CarValue*?
A8. Customers in their 50s tend to have the largest car values.

```
customers %>%
  group_by(Age) %>%
  summarize(avg_value = mean(CarValue)) %>%
  top_n(5)
## # A tibble: 5 x 2
##     Age avg_value
##   <dbl>     <dbl>
## 1    52  37268.83
## 2    54  36547.44
## 3    56  35451.22
## 4    57  37217.39
## 5    58  35458.57
```

Q9. Which credit card has the highest default rate?
A9. Customers with "Other" credit cards tend to have the highest default rate.

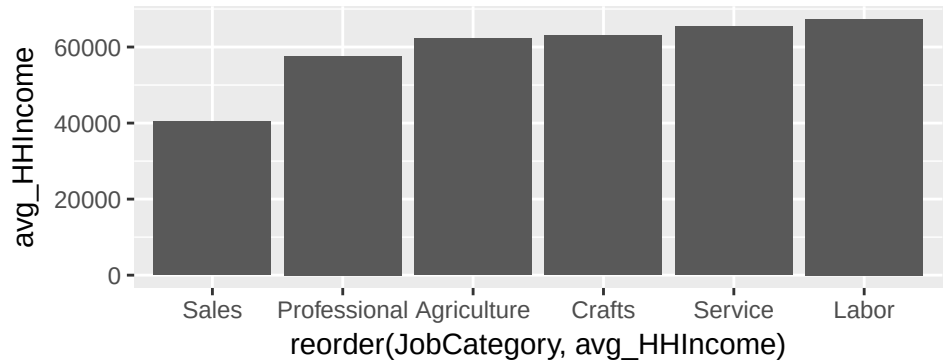
```
customers %>%
  group_by(CreditCard) %>%
  count(LoanDefault) %>%
  mutate(rate = round(n / sum(n), 2)) %>%
  filter(LoanDefault == "Yes") %>%
  arrange(desc(rate))
## Source: local data frame [5 x 4]
## Groups: CreditCard [5]
##
## # A tibble: 5 x 4
##   CreditCard LoanDefault     n  rate
##        <chr>       <chr> <int> <dbl>
## 1       Othe         Yes    64  0.29
## 2       AMEX         Yes   232  0.24
## 3       Disc         Yes   300  0.23
## 4       Visa         Yes   286  0.23
## 5       Mast         Yes   258  0.22
```

Understanting Your Data Through Visualization

We can answer the same questions using base R or `ggplot2` visualizations:

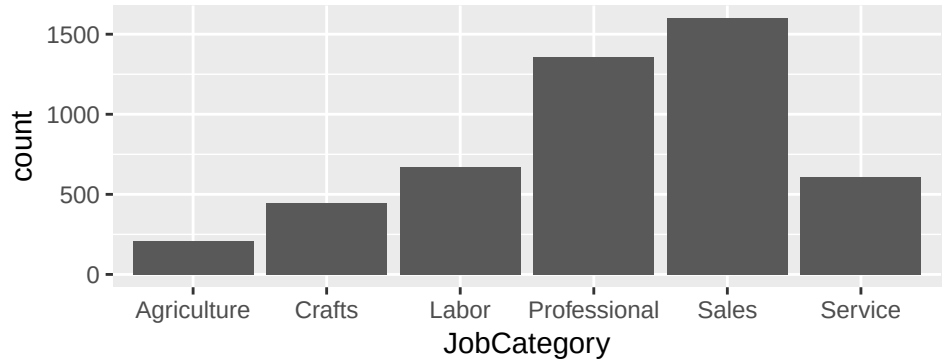
Q1. Which *JobCategory* has the largest average *HHIncome*? Which has the lowest?
A1. We see that the Labor job category has the largest average *HHIncome* while Sales has the lowest.

```
customers %>%
  group_by(JobCategory) %>%
  summarize(avg_HHIncome = mean(HHIncome)) %>%
  ggplot(aes(reorder(JobCategory, avg_HHIncome), avg_HHIncome)) +
  geom_bar(stat = "identity")
```



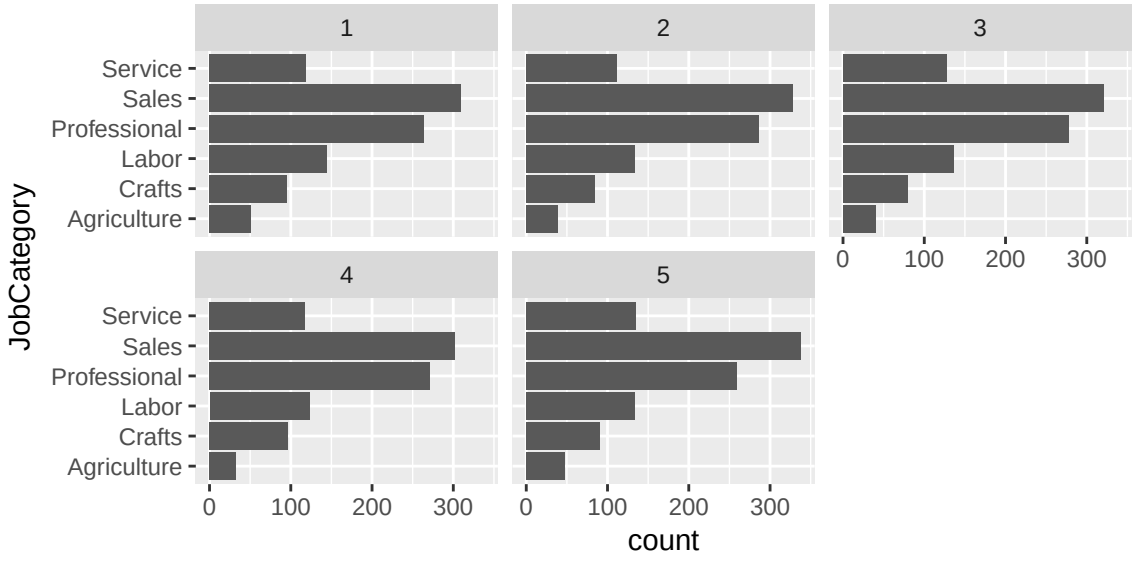
Q2. Is there a large difference in the number of earners in each *JobCategory*?
A2. There is a pretty subtantial difference in the number of earners across the categories.

```
ggplot(customers, aes(JobCategory)) +
  geom_bar()
```



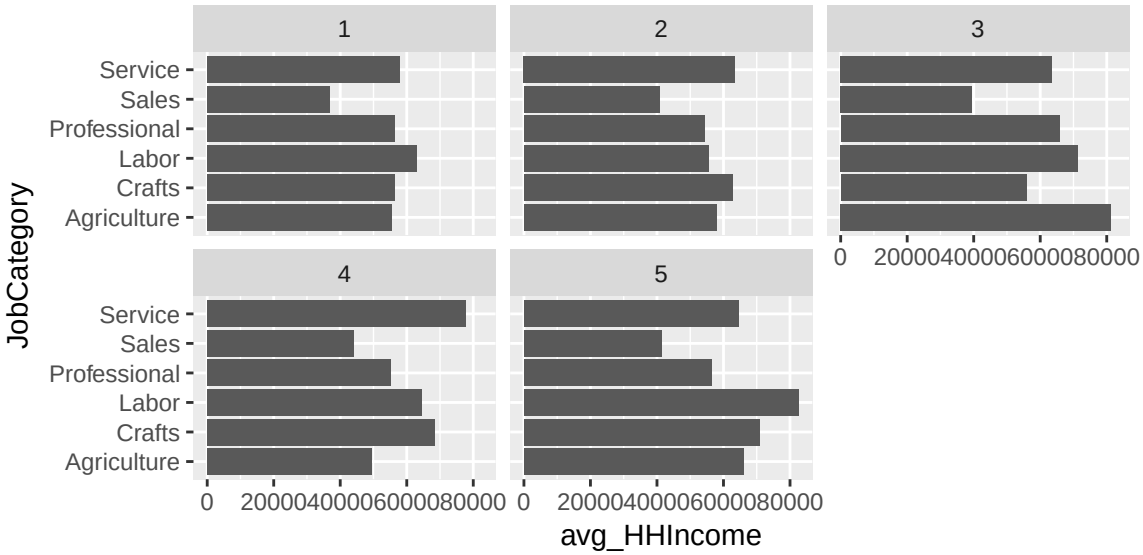
Q3. How many earners are in each *JobCategory* across the regions?
A3. We can inspect the number of earners in each category with the following. They look pretty consistent.

```
ggplot(customers, aes(JobCategory)) +
  geom_bar() +
  facet_wrap(~ Region) +
  coord_flip()
```



Q4. Does the average *HHIncome* vary across *JobCategory* and *Region*?
A4. Yes, there is a fair amount of variation in *HHIncome* across *JobCategory* and less variation across *Region*.

```
customers %>%
  group_by(JobCategory, Region) %>%
  summarize(avg_HHIncome = mean(HHIncome)) %>%
  arrange(desc(avg_HHIncome)) %>%
  ggplot(aes(JobCategory, avg_HHIncome)) +
  geom_bar(stat = "identity") +
  facet_wrap(~ Region) +
  coord_flip()
```

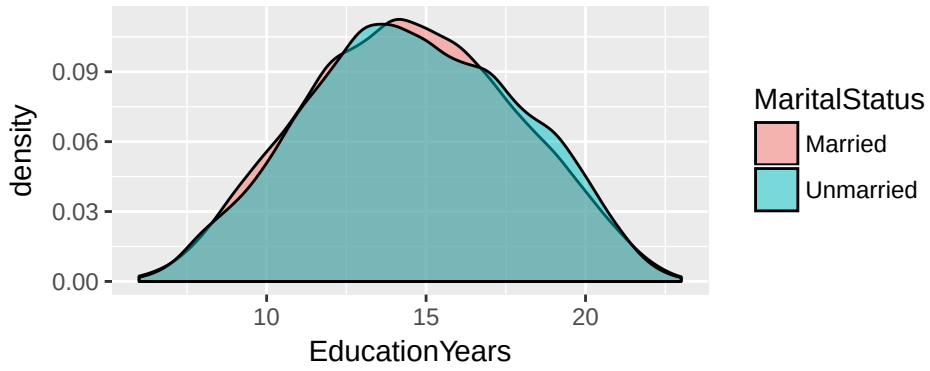


Q5. Do these customers tend to be married, single, or an equal mix?
A5. There are slightly more unmarried customers in this data.

```
customers %>%
  count(MaritalStatus) %>%
  mutate(pct = n / sum(n)) %>%
  ggplot(aes(MaritalStatus, pct)) +
  geom_bar(stat = "identity")
```

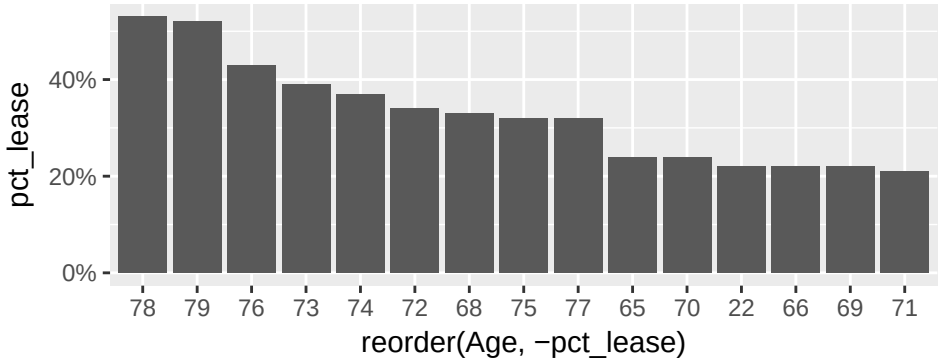
Q6. Are married customers more educated than single customers?
A6. Average education levels appears to be fairly consistent across marital status.

```
ggplot(customers, aes(EducationYears, fill = MaritalStatus)) +
  geom_density(alpha = .5)
```



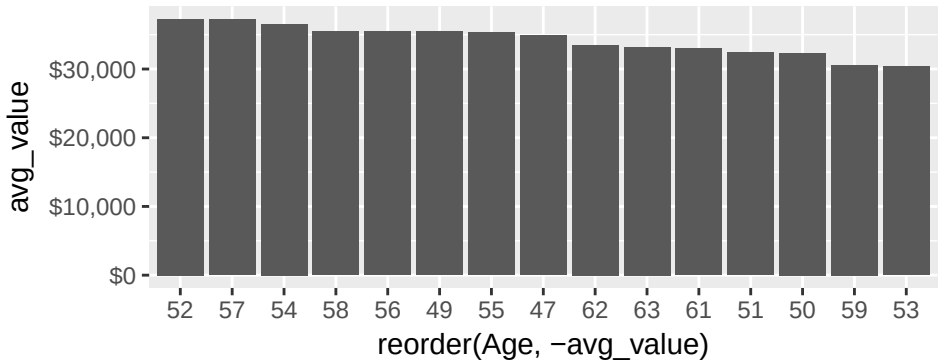
Q7. Which age group is most likely to lease an automobile?
A7. It appears that older customers have a higher percentage of leasing activity than owning or not having an automobile.

```
customers %>%
  group_by(Age) %>%
  count(CarOwnership) %>%
  mutate(pct_lease = round(n / sum(n), 2)) %>%
  filter(CarOwnership == "Lease") %>%
  ungroup() %>%
  top_n(15, wt = pct_lease) %>%
  ggplot(aes(reorder(Age, -pct_lease), pct_lease)) +
  geom_bar(stat = "identity") +
  scale_y_continuous(labels = scales::percent)
```



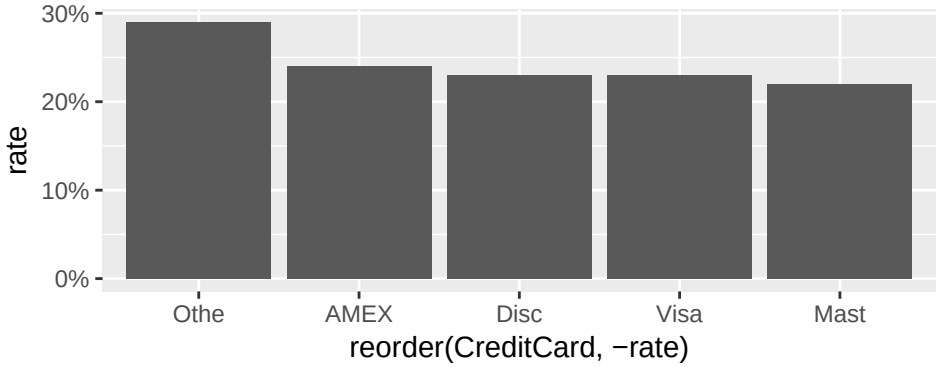
Q8. Which age group tends to have the greatest *CarValue*?
A8. Customers in their 50s tend to have the largest car values.

```
customers %>%
  group_by(Age) %>%
  summarize(avg_value = mean(CarValue)) %>%
  top_n(15) %>%
  ggplot(aes(reorder(Age, -avg_value), avg_value)) +
  geom_bar(stat = "identity") +
  scale_y_continuous(labels = scales::dollar)
```



Q9. Which credit card has the highest default rate?
A9. Customers with "Other" credit cards tend to have the highest default rate.

```
customers %>%
  group_by(CreditCard) %>%
  count(LoanDefault) %>%
  mutate(rate = round(n / sum(n), 2)) %>%
  filter(LoanDefault == "Yes") %>%
  ggplot(aes(reorder(CreditCard, -rate), rate)) +
  geom_bar(stat = "identity") +
  scale_y_continuous(labels = scales::percent)
```

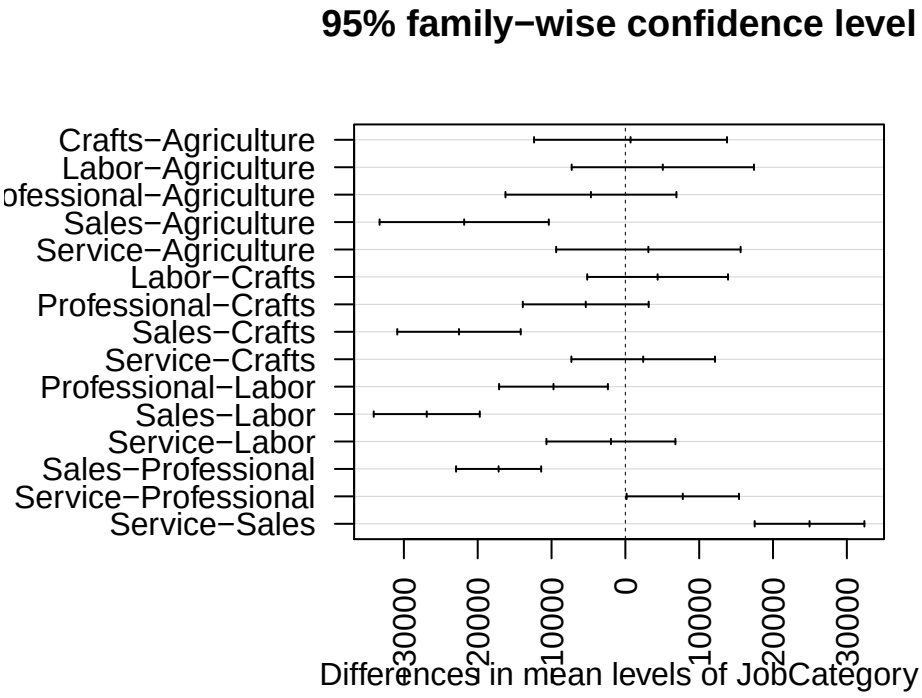


Perform Basic Statistics Inference

Using a p-value < 0.05 to signify statistical significance:

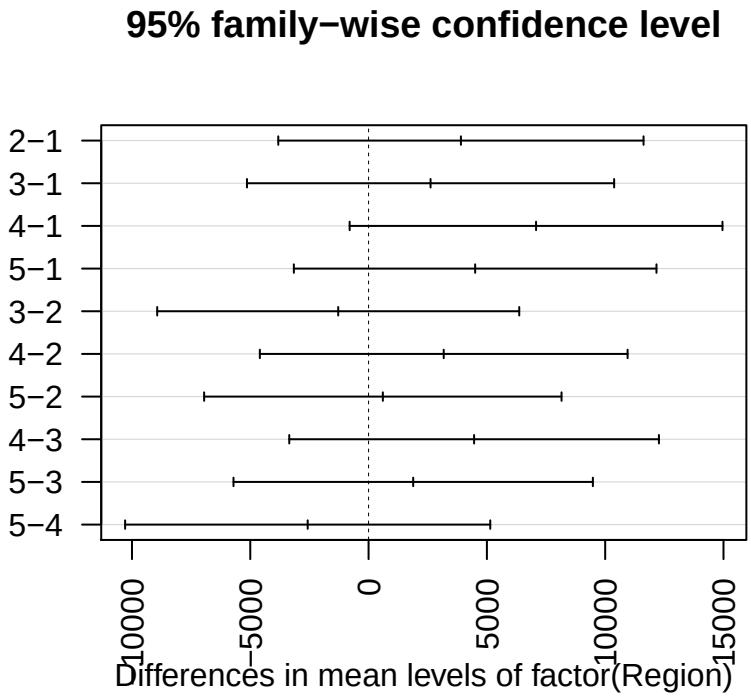
Q1. Does the average *HHIncome* differ by *JobCategory*?
A1. Yes. It appears that Sales tends to have lower *HHIncome* than most other job categories.

```
anova1 <- aov(HHIncome ~ JobCategory, data = customers)
par(oma=c(0,5,0,0))
plot(TukeyHSD(anova1), las = 2)
```



Q2. Does the average *HHIncome* for the sales *JobCategory* differ by *Region*?
A2. No, there is not a statistical difference.

```
sales <- customers %>%
  filter(JobCategory == "Sales")

anova2 <- aov(HHIncome ~ factor(Region), data = sales)
par(oma=c(0,2,0,0))
plot(TukeyHSD(anova2), las = 2)
```

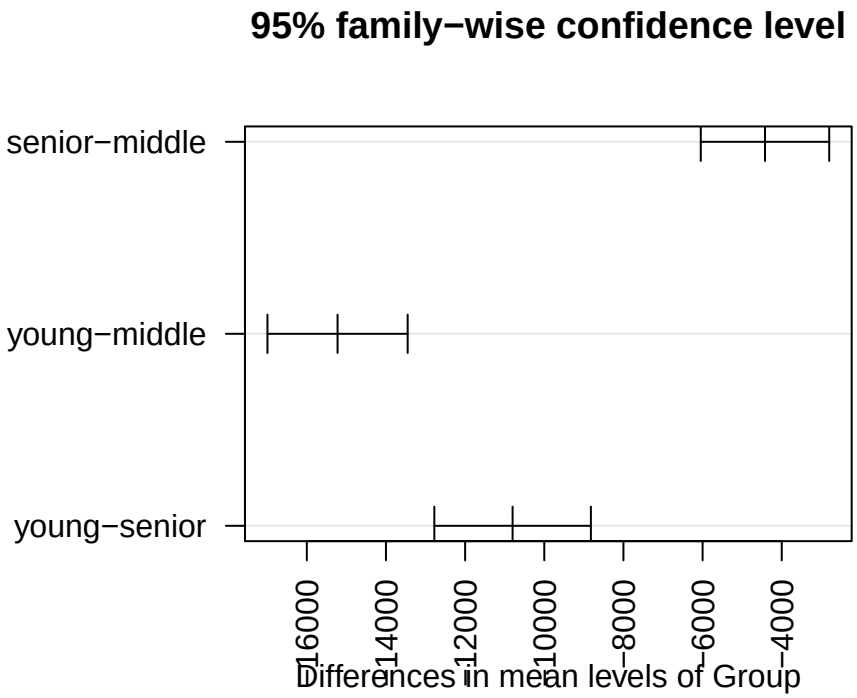


Q3. Are married customers more educated than non-married customers?
A3. No, there is not a statistical difference.

```
t.test(EducationYears ~ MaritalStatus, data = customers)
##
##  Welch Two Sample t-test
##
## data:  EducationYears by MaritalStatus
## t = -1.3691, df = 4864.4, p-value = 0.171
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.31264958  0.05552986
## sample estimates:
##   mean in group Married mean in group Unmarried
##                14.48637                14.61493
```

Q4. Does the average *CarValue* differ across the following categories (young < 30, middle 30-59, senior > 60)?
A4. Yes. Middle aged customers tend to have the largest car value. Seniors tend to have cars valued at $4,423 less than middle aged customers and young customers tend to have cars valued at $15,223 less than middle aged customers.

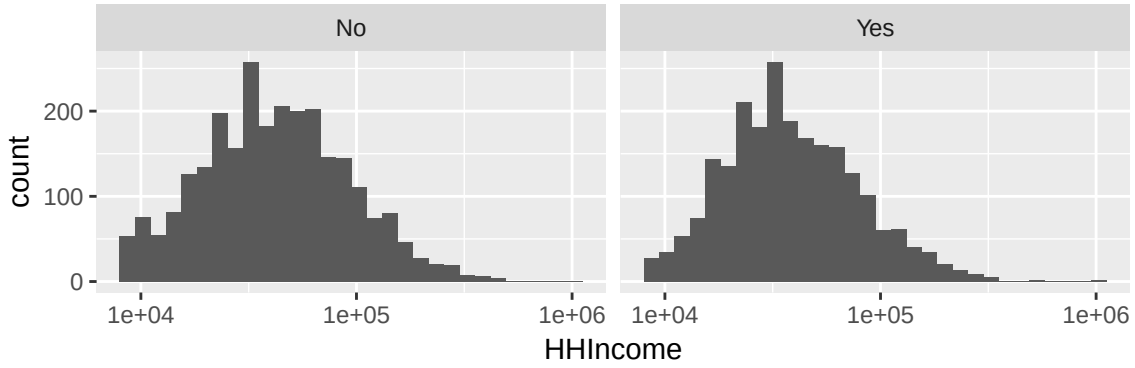
```
age_grp <- customers %>%
  mutate(Group = ifelse(Age < 30, "young", ifelse(Age > 60, "senior", "middle")))

anova3 <- aov(CarValue ~ Group, data = age_grp)
par(oma=c(0,3,0,0))
TukeyHSD(anova3)
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = CarValue ~ Group, data = age_grp)
##
## $Group
##                      diff        lwr        upr p adj
## senior-middle  -4422.994  -6045.383  -2800.605     0
## young-middle  -15223.222 -16994.642 -13451.802     0
## young-senior  -10800.228 -12777.925  -8822.531     0
plot(TukeyHSD(anova3), las = 2)
```



Q5a. Do customers with an *ActiveLifestyle* tend to have higher incomes than inactive customers? Do they tend to have larger *DebtToIncomeRatio*?
A5a. No, their is no statistical difference regardless of how you perform the t.test.

```
t.test(HHIncome ~ ActiveLifestyle, data = customers)
##
##  Welch Two Sample t-test
##
## data:  HHIncome by ActiveLifestyle
## t = 4.1601, df = 4853, p-value = 3.236e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  3497.031 9730.573
## sample estimates:
##  mean in group No mean in group Yes
##          58075.98          51462.18
t.test(log10(HHIncome) ~ ActiveLifestyle, data = customers)
##
##  Welch Two Sample t-test
##
## data:  log10(HHIncome) by ActiveLifestyle
## t = 4.0007, df = 4879.4, p-value = 6.41e-05
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  0.01888114 0.05516707
## sample estimates:
##  mean in group No mean in group Yes
##          4.625219          4.588195
wilcox.test(HHIncome ~ ActiveLifestyle, data = customers)
##
##  Wilcoxon rank sum test with continuity correction
##
```

```
## data:  HHIncome by ActiveLifestyle
## W = 3187400, p-value = 2.115e-05
## alternative hypothesis: true location shift is not equal to 0

ggplot(customers, aes(HHIncome)) +
  geom_histogram() +
  facet_wrap(~ ActiveLifestyle) +
  scale_x_log10()
```

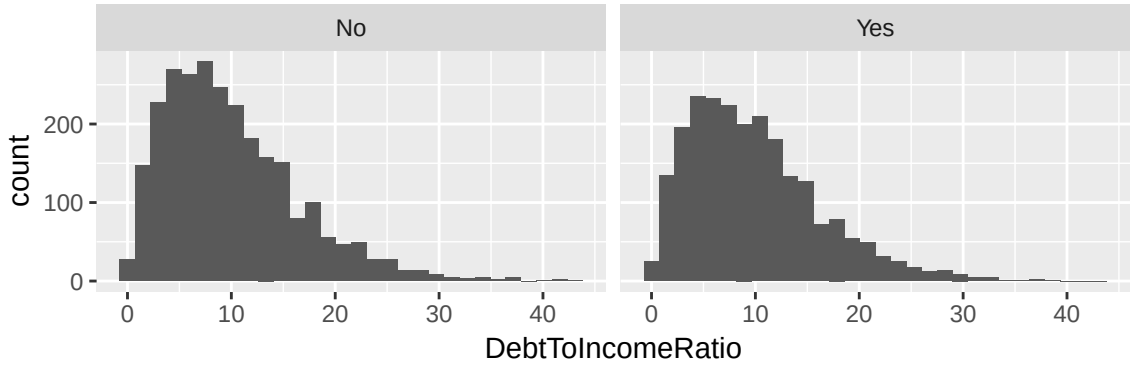


Q5a. Do they tend to have larger *DebtToIncomeRatio*?
A5a. No, their is no statistical difference regardless of how you perform the t.test.

```
t.test(DebtToIncomeRatio ~ ActiveLifestyle, data = customers)
##
##  Welch Two Sample t-test
##
## data:  DebtToIncomeRatio by ActiveLifestyle
## t = 0.48139, df = 4819.1, p-value = 0.6303
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.2701656  0.4460260
## sample estimates:
##  mean in group No mean in group Yes
##          9.965063          9.877133
t.test(log1p(DebtToIncomeRatio) ~ ActiveLifestyle, data = customers)
##
##  Welch Two Sample t-test
##
## data:  log1p(DebtToIncomeRatio) by ActiveLifestyle
## t = 0.5414, df = 4780.2, p-value = 0.5883
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.02574720  0.04539343
## sample estimates:
##  mean in group No mean in group Yes
##          2.214987          2.205164
wilcox.test(DebtToIncomeRatio ~ ActiveLifestyle, data = customers)
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  DebtToIncomeRatio by ActiveLifestyle
## W = 2990900, p-value = 0.7908
```

```
## alternative hypothesis: true location shift is not equal to 0
```

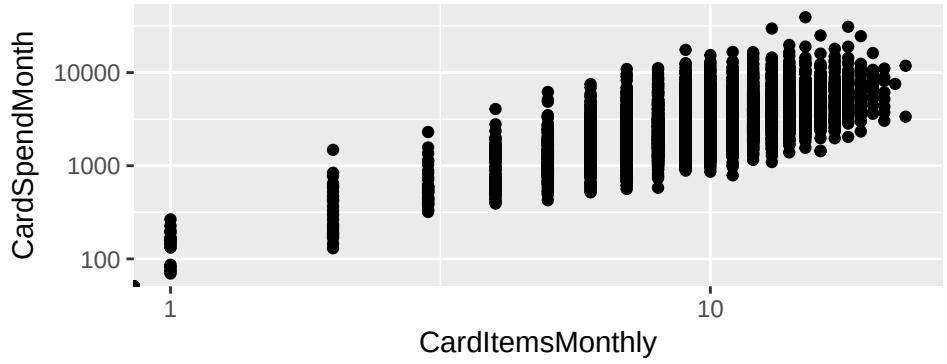
```
ggplot(customers, aes(DebtToIncomeRatio)) +
  geom_histogram() +
  facet_wrap(~ ActiveLifestyle)
```



Identifying Relationships Between Variables

Q1. Is there a strong relationship between *CardItemsMonthly* and *CardSpendMonth* for our customers?
A1. A little investigation shows us that their is a relationship between *CardItemsMonthly* and *CardSpendMonth*. However, this relationship becomes more linear when applying a log-log transformation with a correlation of 0.69.

```
ggplot(customers, aes(CardItemsMonthly, CardSpendMonth)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()
```

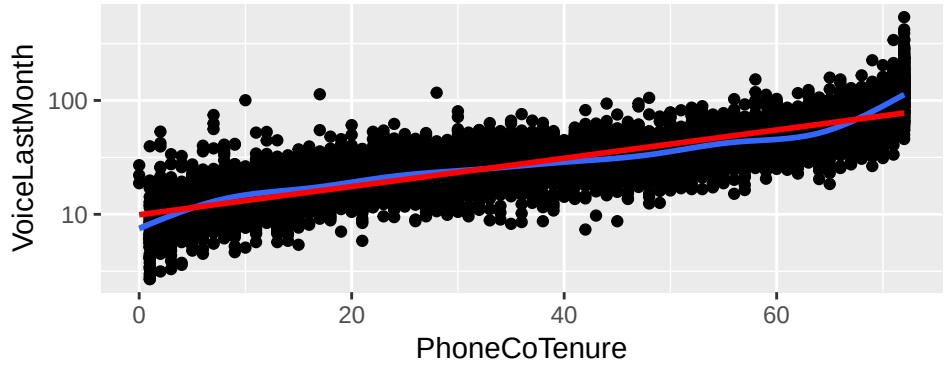



```
customers %>%
  filter(CardItemsMonthly > 0 | CardSpendMonth > 0) %>%
  summarize(r_reg = cor(CardItemsMonthly, CardSpendMonth),
            r_log = cor(log10(CardItemsMonthly), log10(CardSpendMonth)))
## # A tibble: 1 x 2
##       r_reg     r_log
##       <dbl>     <dbl>
## 1 0.5050527 0.6920158
```

Q2. Is there a strong relationship between *PhoneCoTenure* and *VoiceLastMonth* for our customers?
A2. Their is a relationship between *PhoneCoTenure* and *VoiceLastMonth*. This relationship becomes close to linear when applying a log transformation to *VoiceLastMonth* with a correlation of 0.84. However, even

after a log transformation there is still some non-linearity at the two ends of the relationship sprectrum.

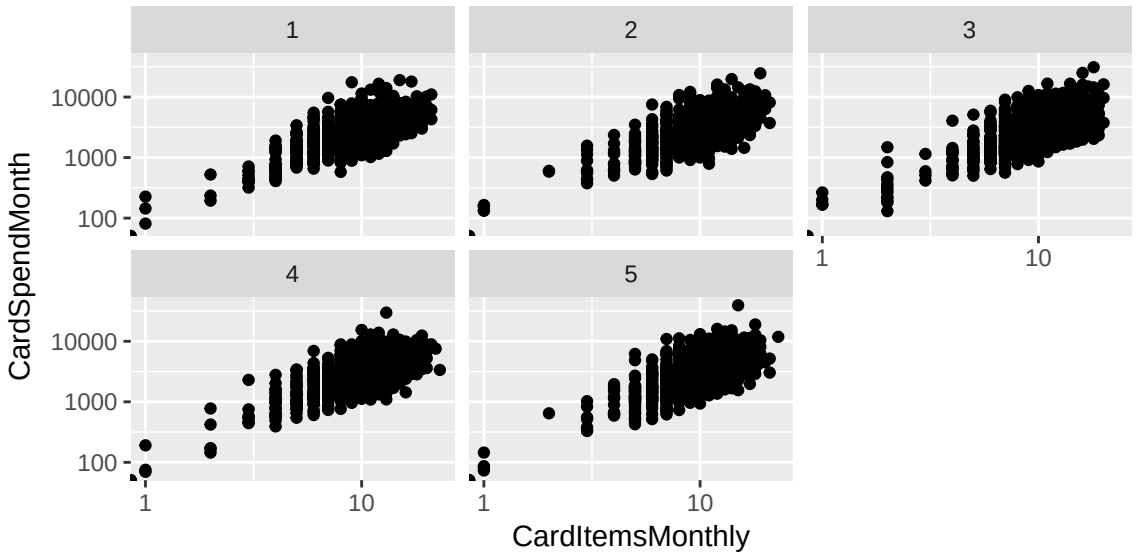
```
ggplot(customers, aes(PhoneCoTenure, VoiceLastMonth )) +
  geom_point() +
  scale_y_log10() +
  geom_smooth() +
  geom_smooth(method = "lm", color = "red")
```




```
customers %>%
  filter(PhoneCoTenure > 0 | VoiceLastMonth > 0) %>%
  summarize(r_reg = cor(PhoneCoTenure, VoiceLastMonth),
            r_log = cor(PhoneCoTenure, log10(VoiceLastMonth)))
## # A tibble: 1 x 2
##      r_reg     r_log
##      <dbl>     <dbl>
## 1 0.670557 0.8353592
```

Q3. Do either of these relationships differ based on region, gender, or credit card type? Do you find this interesting?
A3. The following illustrates how the relationship between *CardItemsMonthly* and *CardSpendMonth* is pretty consistent across the regions. If you perform the same assement for gender and credit card type you will find similiar results. This also occurs with the relationship between *PhoneCoTenure* and *VoiceLastMonth*.

```
ggplot(customers, aes(CardItemsMonthly, CardSpendMonth)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10() +
  facet_wrap(~ Region)
```




```
customers %>%
  filter(CardItemsMonthly > 0 | CardSpendMonth > 0) %>%
  group_by(Region) %>%
  summarize(r_reg = cor(CardItemsMonthly, CardSpendMonth),
            r_log = cor(log10(CardItemsMonthly), log10(CardSpendMonth)))
## # A tibble: 5 x 3
##   Region     r_reg     r_log
##    <dbl>     <dbl>     <dbl>
## 1      1 0.5383714 0.7158806
## 2      2 0.5281559 0.6881467
## 3      3 0.4940950 0.6835311
## 4      4 0.5009015 0.6932835
## 5      5 0.4778773 0.6807617
```

Q4a. Build a simple bivariate regression model to model *CardSpendMonth* as a function of *CardItemsMonthly*?

A4a. Here we remove customers with no expenditures, create a training and test data set, and fit the bivariate regression model.

```
m1_data <- customers %>%
  filter(CardItemsMonthly > 0 | CardSpendMonth > 0)

set.seed(123)
sample <- sample(c(TRUE, FALSE), nrow(m1_data), replace = T, prob = c(0.6, 0.4))
m1_train <- m1_data[sample, ]
m1_test <- m1_data[!sample, ]

m1a <- lm(CardSpendMonth ~ CardItemsMonthly, data = m1_train)
summary(m1a)
##
## Call:
## lm(formula = CardSpendMonth ~ CardItemsMonthly, data = m1_train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4839.1 -1216.6  -457.2   580.6 25248.1
```

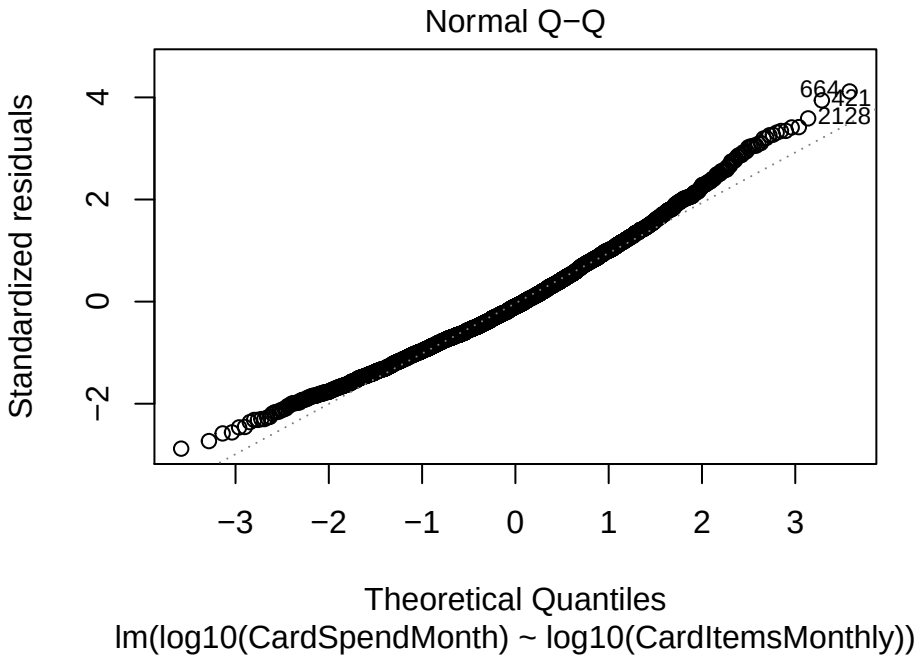
```
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)       -436.52     126.48  -3.451 0.000566 ***
## CardItemsMonthly   375.56      11.73  32.006  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2139 on 2939 degrees of freedom
## Multiple R-squared:  0.2585, Adjusted R-squared:  0.2582
## F-statistic:  1024 on 1 and 2939 DF,  p-value: < 2.2e-16
```

Q4b. Do the residuals suggest this model meets required assumptions? **A4b**. You can assess the residuals by going through the following plots. You will notice that the Q-Q plot illustrates some concerns with normality of our resdiuals.

```
plot(m1a, which = 1)
plot(m1a, which = 2)
plot(m1a, which = 3)
plot(m1a, which = 4)
plot(m1a, which = 5)
```

Q4c. Does a log-log transformation of this model improve the residual fit?
A4c. Fitting a log-log transformation does help improve the residual fit, although slight deviations still occur at the far ends.

```
m1b <- lm(log10(CardSpendMonth) ~ log10(CardItemsMonthly), data = m1_train)
plot(m1b, which = 2)
```



Q4d. What is the out-of-sample MSE for this model?
A4d. We can compute the out-of-sample MSE with the following. First we add the prediction values, then we need to transform our predicted values back since the model applies a log transformation, and then we compute the MSE value.

```
m1_test %>%
  select(CardSpendMonth, CardItemsMonthly) %>%
  add_predictions(m1b) %>%
  mutate(pred_trans = 10^pred) %>%
  summarise(MSE = mean((CardSpendMonth - pred_trans)^2))
## # A tibble: 1 x 1
##       MSE
##     <dbl>
## 1 4587561
```

Q5a. Build a multivariate regression model with *CardSpendMonth* as a function of *HHIncome*, *CreditCard*, *gender*, *region*, and *CardTenure*.
A5a. Here, we fit our multivariate regression model.

```
m2a <- lm(CardSpendMonth ~ HHIncome + CreditCard + Gender + Region + CardTenure,
         data = m1_train)
```

Q5b. Does transforming (i.e. log, sqrt) any of these variables improve the residual diagnostics?
A5b. After some investigation you would find that applying a log-log transformation to *CardSpendMonth* and *HHIncome* does improve the residual diagnostics.

```
m2b <- lm(log1p(CardSpendMonth) ~ log1p(HHIncome) + CreditCard + Gender + Region + CardTenure,
          data = m1_train)
```

Q5c. Which variables are considered statistically significant?
A5c. Looking at the coefficients and their respective p-values we see that nearly all are statistically significant (p-value $< .05$). Region appears to be the one variable that does not meet the statistically significant threshold.

```
summary(m2b)
##
## Call:
## lm(formula = log1p(CardSpendMonth) ~ log1p(HHIncome) + CreditCard +
##     Gender + Region + CardTenure, data = m1_train)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.2748 -0.3460  0.0264  0.3767  2.0049
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      4.8357314  0.1635741  29.563  < 2e-16 ***
## log1p(HHIncome)  0.3267296  0.0152468  21.429  < 2e-16 ***
## CreditCardDisc  -0.5186872  0.0322888 -16.064  < 2e-16 ***
## CreditCardMast  -0.5487957  0.0330376 -16.611  < 2e-16 ***
## CreditCardOthe  -0.4308953  0.0565409  -7.621 3.38e-14 ***
## CreditCardVisa  -0.5083944  0.0326409 -15.575  < 2e-16 ***
## GenderMale       0.0694894  0.0217471   3.195  0.00141 **
## Region           0.0138454  0.0076170   1.818  0.06921 .
## CardTenure      -0.0023187  0.0009166  -2.530  0.01147 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.5865 on 2932 degrees of freedom
## Multiple R-squared:  0.2385, Adjusted R-squared:  0.2364
## F-statistic: 114.8 on 8 and 2932 DF,  p-value: < 2.2e-16
```

Q5d. How would you interpret the coefficients for the statistically significant variables?
A5d. The following are the interpretations:

- *log1p(HHIncome)*: assuming all else equal, for every 1% increase in *HHIncome* there is approximately a 0.33% increase in *CardSpendMonth*.
- *CreditCardDisc*: assuming all else equal, customers with a Discovery credit card have a 0.52% lower *CardSpendMonth* than customers with AMEX credit cards (the baseline).
- *CreditCardMast*: assuming all else equal, customers with a Mastercard credit card have a 0.55% lower *CardSpendMonth* than customers with AMEX credit cards (the baseline).
- *CreditCardOthe*: assuming all else equal, customers with an "Other" credit card have a 0.43% lower *CardSpendMonth* than customers with AMEX credit cards (the baseline).
- *CreditCardVisa*: assuming all else equal, customers with a Visa credit card have a 0.51% lower *CardSpendMonth* than customers with AMEX credit cards (the baseline).
- *GenderMale*: assuming all else equal, males customers have a 0.07% higher *CardSpendMonth* than female customers (the baseline).
- *CardTenure*: assuming all things equal, for every 1 additional tenure year there is a 0.002% decrease in *CardSpendMonth*.

Q5e. What is the out-of-sample MSE for this model?
A5e. The out-of-sample MSE for this model is 4,597,898.

```
m1_test %>%
  select(CardSpendMonth, HHIncome, CreditCard, Gender, Region, CardTenure) %>%
  add_predictions(m2b) %>%
  mutate(pred_trans = expm1(pred)) %>%
  summarise(MSE = mean((CardSpendMonth - pred_trans)^2))
## # A tibble: 1 x 1
##       MSE
##     <dbl>
## 1 4597898
```