Data Visualization and Exploration with R

A practical guide to using R, RStudio, and Tidyverse for data visualization, exploration, and data science applications.

GeoSpatial Training Services

Eric Pimpler

Data Visualization and Exploration with R

A practical guide to using R, RStudio, and Tidyverse for data visualization, exploration, and data science applications.

Eric Pimpler

Introduction to Data Visualization and Exploration with R

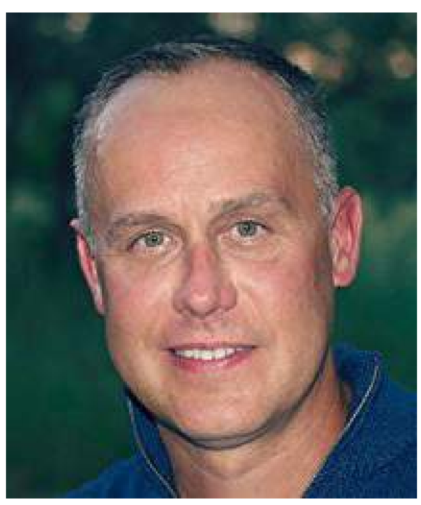
A practical guide to using R, RStudio, and tidyverse for data visualization, exploration, and data science applications.
Eric Pimpler

Geospatial Training Services 215 W Bandera #114-104
Boerne, TX 78006
PH: 210-260-4992
Email: sales@geospatialtraining.com Web: http://geospatialtraining.com Twitter: @gistraining

Copyright © 2017 by Eric Pimpler – Geospatial Training Services All rights reserved.

No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without written permission from the author, except for the use of brief quotations in a book review.

About the Author



Eric Pimpler

Eric Pimpler is the founder and
owner of Geospatial Training Services (geospatialtraining.com) and have over 25 years of experience implementing and teaching GIS solutions using Esri software. Currently he focuses on data science applications with R along with ArcGIS Pro and Desktop scripting with Python and the development of custom ArcGIS Enterprise (Server) and ArcGIS Online web and mobile applications with JavaScript.

Eric is the also the author of several other books including *Introduction to Programming ArcGIS Pro with Python (https://www.amazon.com/dp/1979451079/re (https://www.amazon.com/dp/1979451079/re 1&keywords=Programming+ArcGIS+Pro+with +Python), Programming ArcGIS with Python Cookbook* (https://www.packtpub.com/ application-development/programmingarcgis-python-cookbook-second-edition), *Spatial Analytics with ArcGIS* (https://www. packtpub.com/application-development/ spatial-analytics-arcgis), *Building Web and Mobile ArcGIS Server Applications with JavaScript* (https://www.packtpub.com/ application-development/building-weband-mobile-arcgis-server-applicationsjavascript), and *ArcGIS Blueprints* (https:// www.packtpub.com/application development/arcgis-blueprints).

If you need consulting assistance with your data science or GIS projets please contact Eric at eric@geospatialtraining. com or sales@geospatialtraining.com. Geospatial Training Services provides contract application development and programming expertise for R, ArcGIS Pro, ArcGIS Desktop, ArcGIS Enterprise (Server), and ArcGIS Online using Python, .NET/ArcObjects, and JavaScript.

Downloading and Installing Exercise Data for this Book

This is intended as a hands-on exercise book and is designed to give you as much handson coding experience with R as possible. Many of the exercises in this book require that you load data from a file-based data source such as a CSV file. These files will need to be installed on your computer before continuing with the exercises in this chapter as well as the rest of the book. Please follow the instructions below to download and install the exercise data

1. In a web browser go to one of the links below to download the exercise data: https://www.dropbox.com/s/5p7j7nl8hgijsnx/IntroR.zip?dl=0.

https://s3.amazonaws.com/VirtualGISClassroom/IntroR/IntroR.zip
2. This will download a file called IntroR.zip.

3. The exercise data can be unzipped to any location on your computer. After unzipping the IntroR.zip file you will have a folder structure that includes IntroR as the top-most folder with sub-folders called Data and Solutions. The Data folder contains the data that will be used in the exercises in the book, while the Solutions folder contains solution files for the R script that you will write.

RStudio can be used on Windows, Mac, or Linux so rather than specifying a specific folder to place the data I will leave the installation location up to you. Just remember where you unzip the data because you'll need to reference the location when you set the working directory.

4. For reference purposes I have installed the data to the desktop of my Mac computer under IntroR\Data. You will see this location referenced at various locations throughout the book. However, keep in mind that you can install the data anywhere.

Table of Contents

CHAPTER 1: Introduction to R and RStudio 9

Introduction to RStudio 10
Exercise 1: Creating variables and assigning data 27
Exercise 2: Using vectors and factors 32
Exercise 3: Using lists 36
Exercise 4: Using data classes 39
Exercise 5: Looping statements 46
Exercise 6: Decision support statements – if | else 48
Exercise 7: Using functions 51
Exercise 8: Introduction to tidyverse 53

CHAPTER 2: The Basics of Data Exploration and Visualization with R 57

Exercise 1: Installing and loading tidyverse 58
Exercise 2: Loading and examining a dataset...60
Exercise 3: Filtering a dataset 64
Exercise 4: Grouping and summarizing a dataset 65
Exercise 5: Plotting a dataset 66
Exercise 6: Graphing burglaries by month and year

....67

CHAPTER 3: Loading Data into R
.... 73

Exercise 1: Loading a csv file with read.table()
....73
Exercise 2: Loading a csv file with read.csv()
....76
Exercise 3: Loading a tab delimited file with read.table()
....77
Exercise 4: Using readr to load data
....77

CHAPTER 4: Transforming Data
.... 83

Exercise 1: Filtering records to create a subset
....84
Exercise 2: Narrowing the list of columns with select()
....87
Exercise 3: Arranging Rows
....90
Exercise 4: Adding Rows with mutate()
....92
Exercise 5: Summarizing and Grouping
....94
Exercise 6: Piping
....97
Exercise 7: Challenge
....99

CHAPTER 5: Creating Tidy Data 101

Exercise 1: Gathering
....102
Exercise 2: Spreading
....107

Exercise 3: Separating
....110
Exercise 4: Uniting
....113

CHAPTER 6: Basic Data Exploration Techniques in R 115

Exercise 1: Measuring Categorical Variation with a Bar Chart
....116
Exercise 2: Measuring Continuous Variation with a Histogram
....118
Exercise 3: Measuring Covariation with Box Plots
....120
Exercise 4: Measuring Covariation with Symbol Size
....122
Exercise 5: 2D bin and hex charts
....124
Exercise 6: Generating Summary Statistics
....126

CHAPTER 7: Basic Data Visualization Techniques 129

Step 1: Creating a scatterplot
....130
Step 2: Adding a regression line to the scatterplot
....133
Step 3: Plotting categories
....136
Step 4: Labeling the graph
....137
Step 5: Legend layouts
....144
Step 6: Creating a facet
....146
Step 7:
Theming...
Step 8: Creating bar charts

....148
Step 9: Creating Violin Plots
....150
Step 10: Creating density plots
....153

CHAPTER 8: Visualizing Geographic Data with ggmap 157

Exercise 1: Creating a basemap
....158
Exercise 2: Adding operational data layers
....162
Exercise 3: Adding Layers from Shapefiles
....169

CHAPTER 9: R Markdown 173

Exercise 1: Creating an R Markdown file
....175
Exercise 2: Adding Code Chunks and Text to an R Markdown File
................................178
Exercise 3: Code chunk and header options
....190
Exercise 4: Caching
....199
Exercise 5: Using Knit to output an R Markdown file
....201

CHAPTER 10: Case Study – Wildfire Activity in the Western United States 205

Exercise 1: Have the number of wildfires increased or decreased in the past few decades? 207
Exercise 2: Has the acreage burned increased over time?
....211
Exercise 3: Is the size of individual wildfires increasing over time?
..................................220

Exercise 4: Has the length of the fire season increased over time? 225
Exercise 5: Does the average wildfire size differ by federal organization 230

CHAPTER 11: Case Study – Single Family Residential Home and Rental Values 233

Exercise 1: What is the trend for home values in the Austin metro area 234
Exercise 2: What is the trend for rental rates in the Austin metro area? 240
Exercise 3: Determining the Price-Rent Ratio for the Austin metropolitan area 242
Exercise 4: Comparing residential home values in Austin to other Texas and U.S. metropolitan areas 247

Chapter 1

Introduction to R and RStudio

The R Project for Statistical Computing, or simply named R, is a free software environment for statistical computing and graphics. It is also a programming language that is widely used among statisticians and data miners for developing statistical software and data analysis. Over the last few years, they were joined by enterprises who discovered the potential of R, as well as technology vendors that offer R support or R-based products.

Although there are other programming languages for handling statistics, R has become the de facto language of statistical routines, offering a package repository with over 6400 problem-solving packages. It is also offers versatile and powerful plotting. It also has the advantage of treating tabular and multi-dimensional data as a labeled, indexed series of observations. This is a game changer over typical software which is just doing 2D layout, like Excel.

In this chapter we'll cover the following topics:

- Introduction to RStudio
- Creating variables and assigning data
- Using vectors and factors
- Using lists
- Using data classes
- Looping statements
- Decision support statements
- Using functions
- Introduction to tidyverse

Introduction to RStudio

There are a number of integrated development environments (IDE) that you can use to write R code including Visual Studio for R, Eclipse, R Console, and RStudio among others. You could also use a plain text editor as well. However, we're going to use RStudio for the exercises in this book. RStudio is a free, open source IDE for R. It includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workspace management.

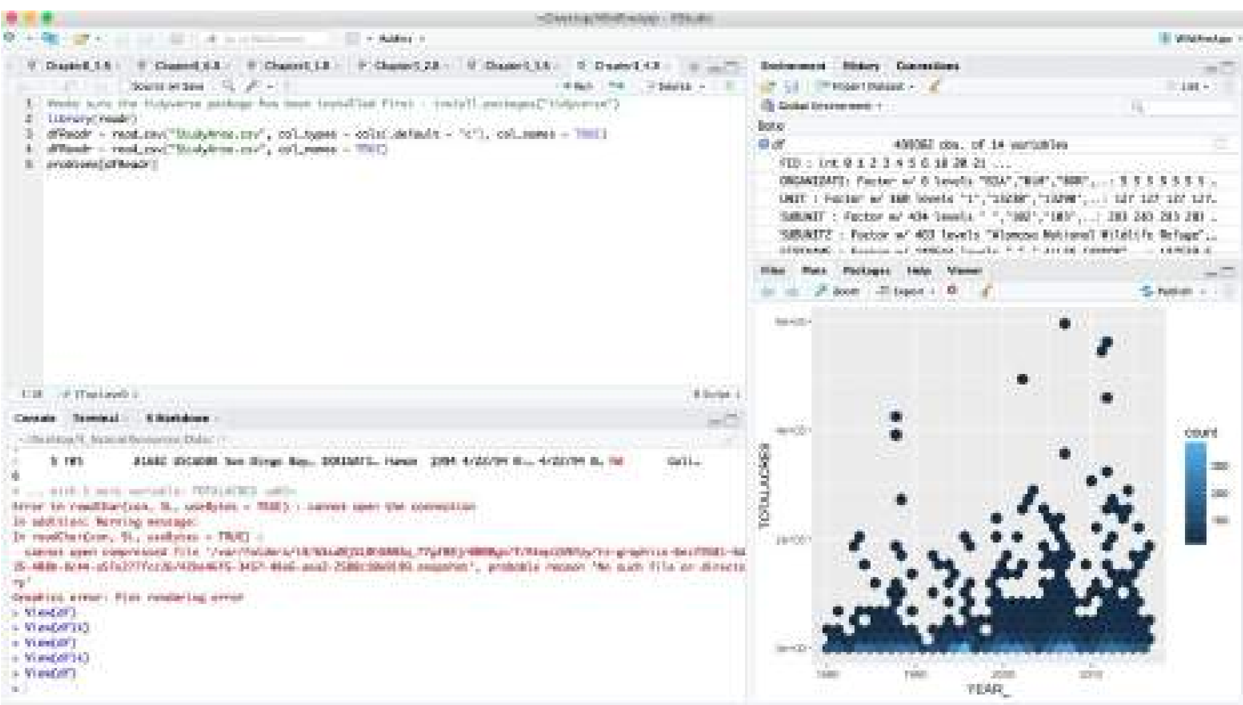
RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux) or in a browser connected to RStudio Server or RStudio Server Pro (Debian/Ubuntu, RedHat/CentOS, and SUSE Linux).

Although there are many options for R development, we're going to use RStudio for the exercises in this book. You can get more information on RStudio at

https://www.rstudio.com/products/rstudio/

The RStudio Interface

The RStudio Interface, displayed in the screenshot below, looks quite complex initially, but when you break the interface down into sections it isn't so overwhelming. We'll cover much of the interface in the sections below. Keep in mind though that the interface is customizable so if you find the default interface isn't exactly what you like it can be changed. You'll learn how to customize the interface in a later section.



To simplify the overview of RStudio we'll break the IDE into quadrants to make it easier to reference each component of the interface. The screenshot below illustrates each of the quadrants. We'll start with the panes in quadrant 1 and work through each of the quadrants.

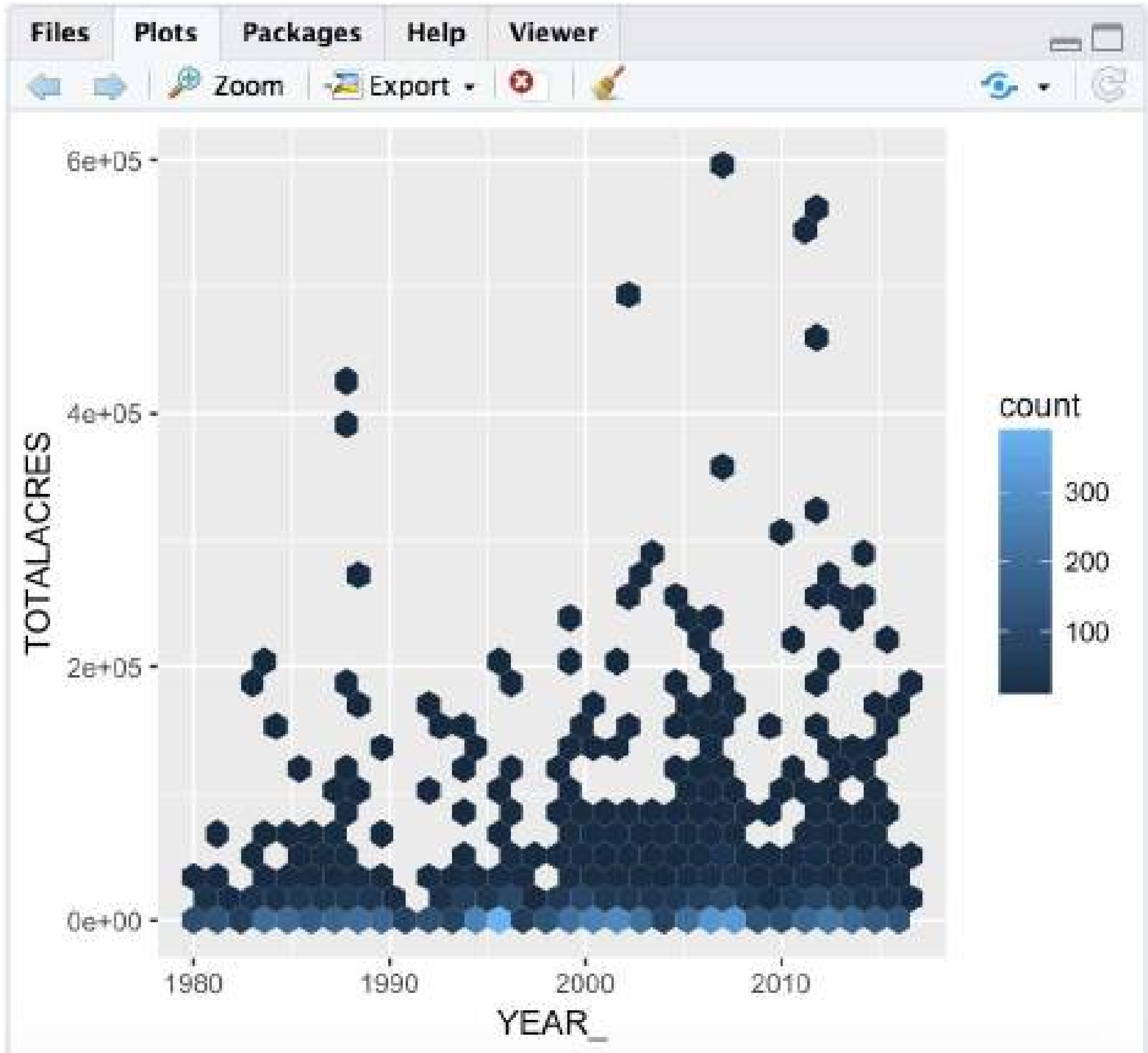
Files Pane – (Q1)

The **Files**pane functions like a file explorer similar to Windows Explorer on a Windows operating system or Finder on a Mac. This tab, displayed in the screenshot below, provides the following functionality:

1. Delete files and folders
2. Create new folders
3. Rename folders
4. Folder navigation
5. Copy or move files
6. Set working directory or go to working directory
7. View files
8. Import datasets

Plots Pane – (Q1)

The **Plots**pane, displayed in the screenshot below, is used to view output visualizations produced when typing code into the **Console** window or running a script. Plots can be created using a variety of different packages, but we'll primarily be using the ggplot2 package in this book. Once produced, you can zoom in, export as an image, or PDF, copy to the clipboard, and remove plots. You can also can navigate to previous and next plots.



Packages Pane – (Q1)

The **Packages** pane, shown in the screenshot below, displays all currently installed packages along with a brief description and version number for the package. Packages can also be removed using the x icon to the right of the version number for the package. Clicking on the package name will display the help file for the package in the **Help** tab. Clicking on the checkbox to the left of the package name loads the library so that it can be used when writing code in the **Console** window.

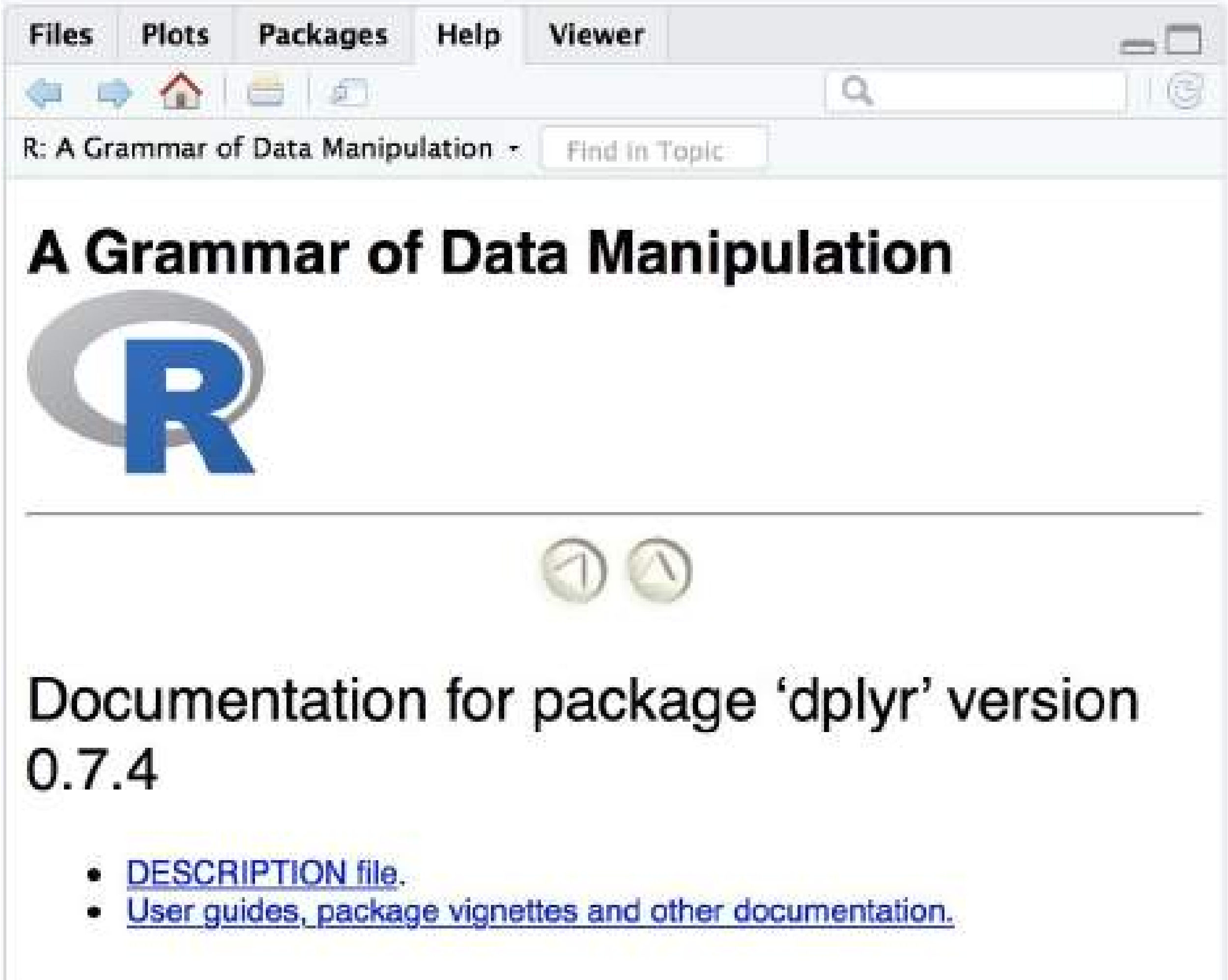
Files | Plots | Packages | Help | Viewer

Install | Update | Packrat

	Name	Description	Version
System Library			
☐	acepack	ACE and AVAS for Selecting Multiple Regression Transformations	1.4.1
☐	assertthat	Easy Pre and Post Assertions	0.2.0
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.2
☐	base64enc	Tools for base64 encoding	0.1-3
☐	BH	Boost C++ Header Files	1.66.0-1
☐	bindr	Parametrized Active Bindings	0.1.1
☑	bindrcpp	An 'Rcpp' Interface to Active Bindings	0.2
☑	bitops	Bitwise Operations	1.0-6
☐	boot	Bootstrap Functions (Originally by Angelo Canty for S)	1.3-20
☐	broom	Convert Statistical Analysis Objects into Tidy Data Frames	0.4.3
☐	callr	Call R from R	2.0.2
☐	caTools	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.17.1
☐	cellranger	Translate Spreadsheet Cell Ranges to Rows and Columns	1.1.0
☐	checkmate	Fast and Versatile Argument Checks	1.8.5

Help Pane – (Q1)

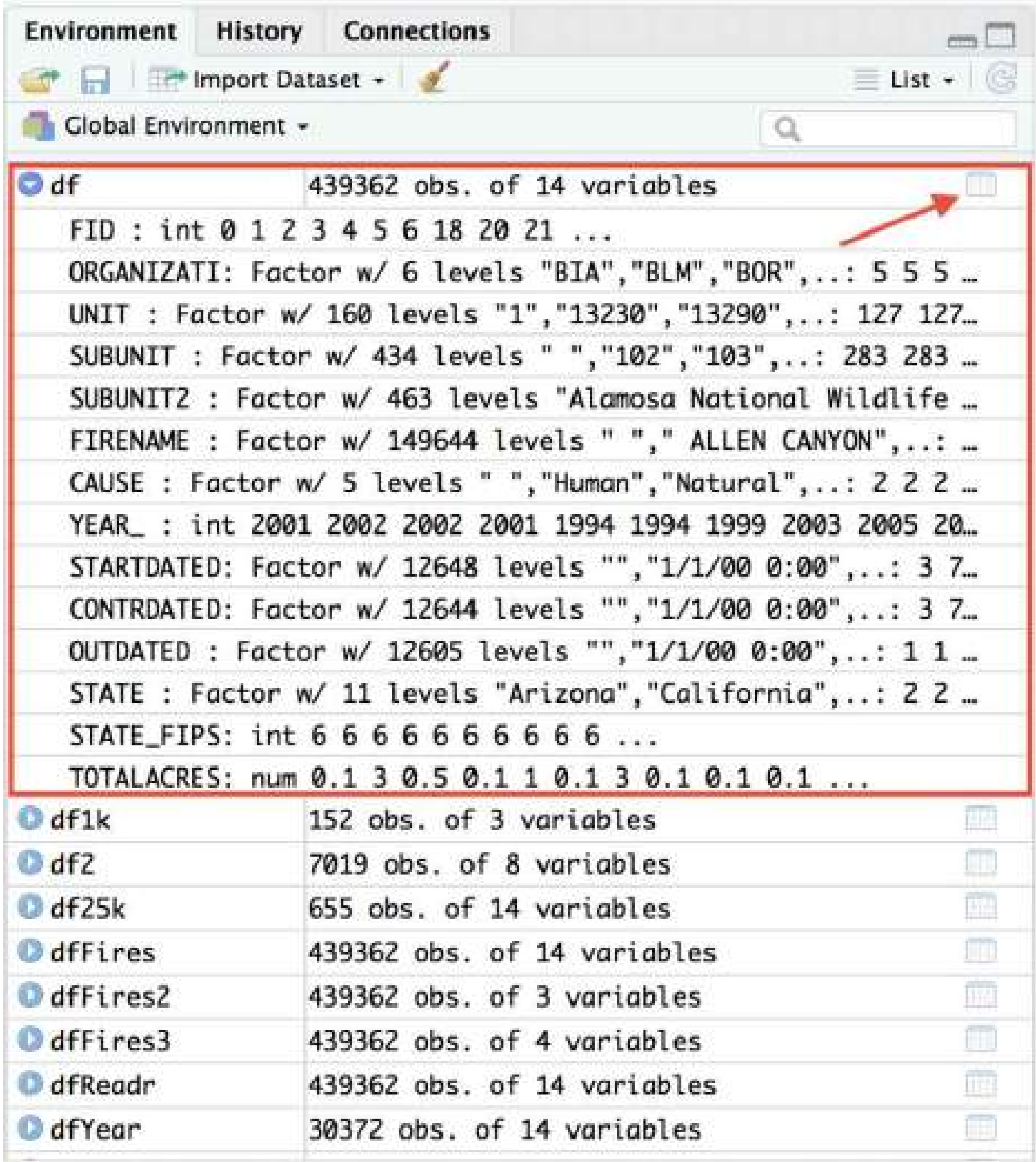
The **Help** pane, shown in the screenshot below, displays linked help documentation for any packages that you have installed.



Viewer Pane – (Q1)
RStudio includes a **Viewer**pane that can be used to view local web content. For example, web graphics generated using packages like googleVis, htmlwidgets, and RCharts, or even a local web application created with Shiny. However, keep in mind that the Viewer pane can only be used for local web content in the form of static HTML pages written in the session's temporary directory or a locally run web application. The Viewer pane can't be used to view online content.

Environment Pane – (Q2)

The **Environment** pane contains a listing of variables that you have created for the current session. Each variable is listed in the tab and can be expanded to view the contents of the variable. You can see an example of this in the screenshot below by taking a look at the df variable. The rectangle surrounding the df variable displays the columns for the variable.



Clicking the table icon on the far-right side of the display (highlighted with the arrow in the screenshot above) will open the data in a tabular viewer as seen in the screenshot below.

Filter

	FID	ORGANIZATI	UNIT	SUBUNIT	SUBUNIT2	FIRENAME	CAUSE	YEAR_	STARTDATED
1	0	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	PUMP HOUSE	Human	2001	1/1/01 0:00
2	1	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	I5	Human	2002	5/3/02 0:00
3	2	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	SOUTHBAY	Human	2002	6/1/02 0:00
4	3	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	MARINA	Human	2001	7/12/01 0:00
5	4	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	HILL	Human	1994	9/13/94 0:00
6	5	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	IRRIGATION	Human	1994	4/22/94 0:00
7	6	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	FIELD	Human	1999	12/6/99 0:00
8	18	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	CALLA FIRE	Human	2003	6/3/03 0:00
9	20	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	OVERPASS	Human	2005	8/20/05 0:00
10	21	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	TRAIN FIRE	Human	2005	12/11/05 0:00
11	22	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	MARSH	Human	2004	1/12/04 0:00
12	23	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	OTAY LAKE #2	Human	2004	4/12/04 0:00
13	24	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	BAYSIDE FIRE	Human	2004	12/5/04 0:00

Showing 1 to 13 of 439,362 entries

Other functionality provided by the **Environment** pane includes opening or saving a workspace, importing dataset from text files, Excel spreadsheets, and various statistical package formats. You can also clear the current workspace.

History Pane – (Q2)

The **History** pane, shown in the screenshot below, displays a list of all commands that have been executed in the current session. This tab includes a number of useful functions including the ability to save these commands to a file or load historical commands from an existing file. You can also select specific commands from the History tab and send them directly to the console or an open script. You can also remove items from the **History** pane.

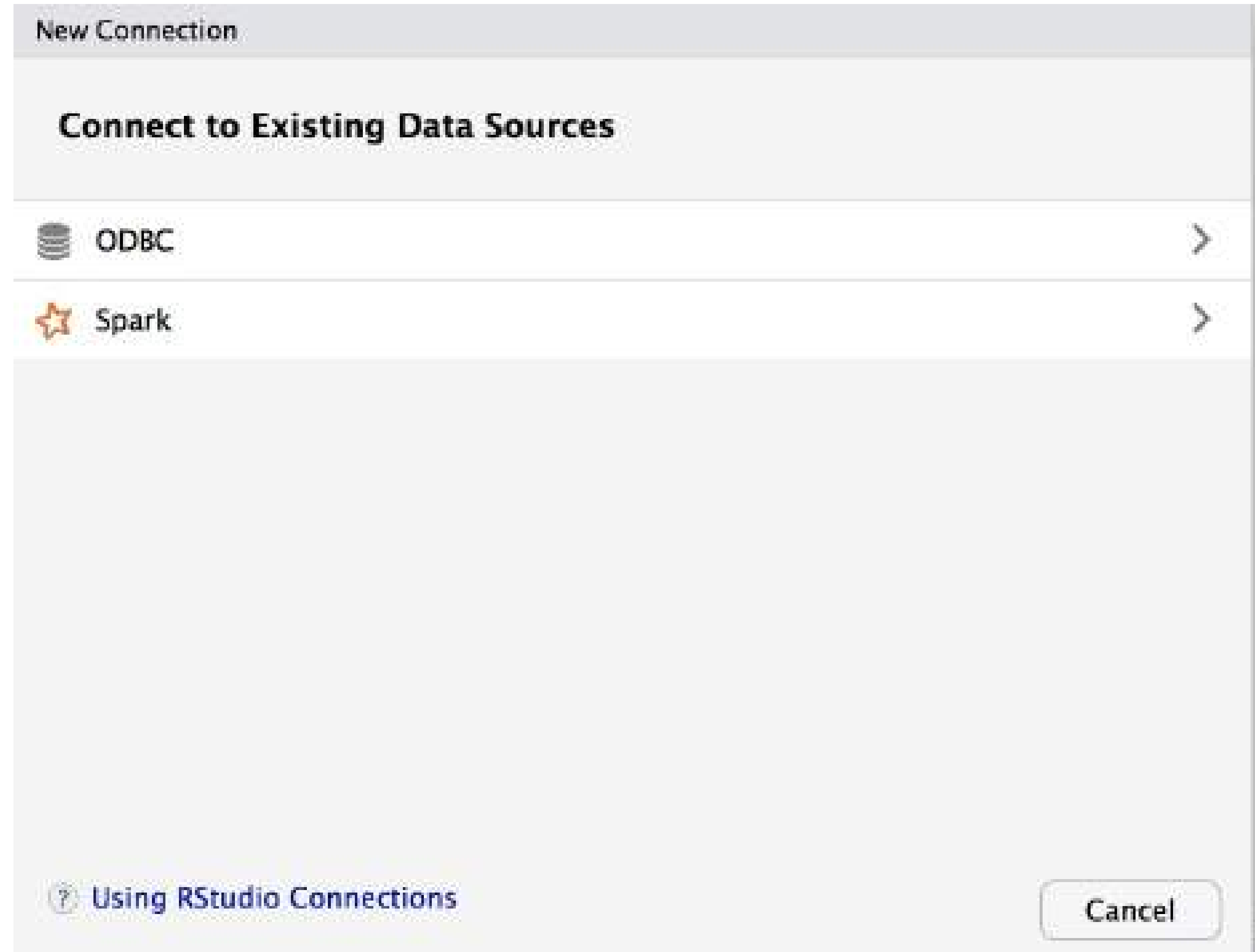
Environment History Connections

To Console To Source

```
df %>%
filter(TOTALACRES >= 1000) %>%
ggplot() + geom_hex(mapping = aes(x=YEAR_, y=TOTALACRES))
source('~/Desktop/R_NaturalResources/Solutions/Chapter8_6.R', echo…
source('~/Desktop/R_NaturalResources/Solutions/Chapter8_6.R', echo…
source('~/Desktop/R_NaturalResources/Solutions/Chapter8_6.R', echo…
df = read_csv("StudyArea.csv", col_types = list(UNIT = col_charact…
source('~/Desktop/R_NaturalResources/Solutions/Chapter8_6.R', echo…
source('~/Desktop/R_NaturalResources/Solutions/Chapter8_6.R', echo…
median(df$TOTALACRES)
summary(df$TOTALACRES)
df = read.table("StudyArea.csv", header = TRUE)
df = read.table("StudyArea.csv", sep=",", header = TRUE)
df = read.table("StudyArea.csv", header=TRUE, fill=TRUE, sep=",")
df = read.table("StudyArea.csv", header=TRUE, fill=TRUE, quote="",…
df = read.csv("StudyArea.csv")
df2 = read.table("all_genes_pombase.txt", header=TRUE, sep="\t", q…
df2 = read.table("all_genes_pombase.txt", header=TRUE, sep="\t", q…
dfReadr = read_csv("StudyArea.csv", col_types = cols(.default = "c…
dfReadr = read_csv("StudyArea.csv", col_names = TRUE)
problems(dfReadr)
dfReadr = read_csv("StudyArea.csv", col_types = list(UNIT = col_ch…
head(dfReadr)
View(df)
View(df1k)
View(df)
```

Connections Pane – (Q2)

The **Connections**tab can be used to access existing or create new connections to ODBC and Spark data sources.



Source Pane – (Q3)

The **Source**pane in RStudio, seen in the screenshot below, is used to create scripts, and display datasets An R script is simply a text file containing a series of commands that are executed together. Commands can also be written line by line from the **Console** pane as well. When written from the **Console**pane, each line of code is executed when you click the Enter (Return) key. However, scripts are executed as a group.

Multiple scripts can be open at the same time with each script occupying a separate tab as seen in the screenshot. RStudio provides the ability to execute the entire script, only the current line, or a highlighted group of lines. This gives you a lot of control over the execution the code in a script.

```
df = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
#steps 1-3
df %>%
  select(ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE) %>%
  filter(TOTALACRES >= 1000) %>%
  ggplot() + geom_histogram(mapping = aes(x=TOTALACRES), binwidth=500)

#step 4
df %>%
  count(cut_width(TOTALACRES, 500))
```

The **Source** pane can also be used to display datasets. In the screenshot below, a data frame is displayed. Data frames can be displayed in this manner by calling the View(<data frame>) function.

	FID	ORGANIZATI	UNIT	SUBUNIT	SUBUNIT2	FIRENAME	CAUSE
1	0	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	PUMP HOUSE	Human
2	1	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	I5	Human
3	2	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	SOUTHBAY	Human
4	3	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	MARINA	Human
5	4	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	HILL	Human
6	5	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	IRRIGATION	Human
7	6	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	FIELD	Human
8	18	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	CALLA FIRE	Human
9	20	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	OVERPASS	Human
10	21	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	TRAIN FIRE	Human
11	22	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	MARSH	Human
12	23	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	OTAY LAKE #2	Human
13	24	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	BAYSIDE FIRE	Human

Showing 1 to 13 of 439,362 entries

Console Pane – (Q4)

The **Console**pane in RStudio is used to interactively write and run lines of code. Each time you enter a line of code and click **Enter**(Return) it will execute that line of code. Any warning or error messages will be displayed in the **Console**

window as well as output from **print()** statements.

```
✔ tidyr   0.8.0     ✔ stringr 1.3.0
✔ ggplot2 2.2.1     ✔ forcats 0.3.0
— Conflicts ————————————————————————————————— tidyverse_conflicts() —
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
> df = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
Warning message:
The following named parsers don't match the column names: UNIT
>
> df = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
Warning message:
The following named parsers don't match the column names: UNIT
> setwd("~/Desktop/R_NaturalResources/Data")
> df = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
> View(df)
>
```

Terminal Pane – (Q4)

The RStudio **Terminal**pane provides access to the system shell from within the RStudio IDE. It supports xterm emulation, enabling use of full-screen terminal applications (e.g. text editors, terminal multiplexers) as well as regular command-line operations with lineediting and shell history.

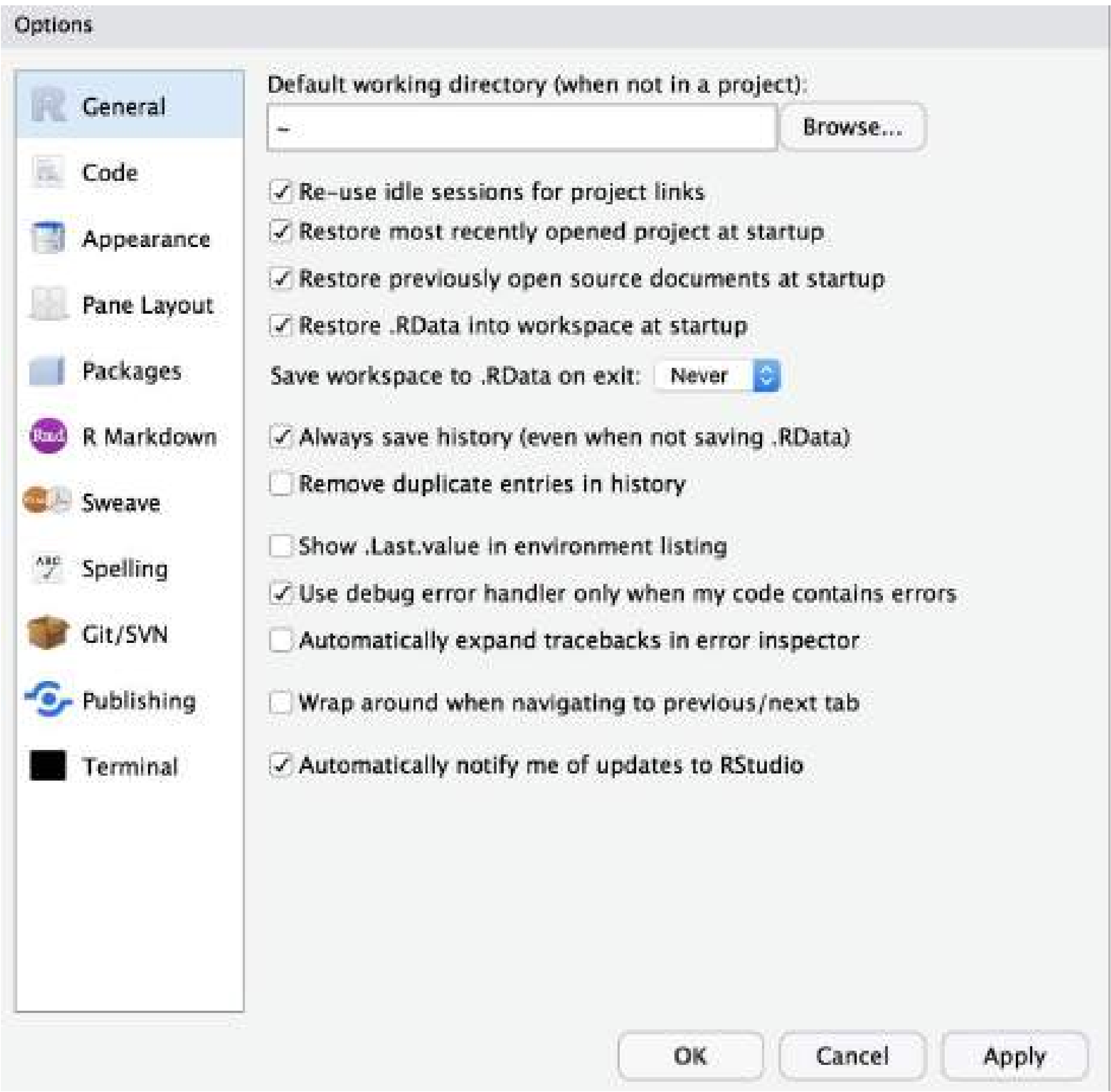
There are many potential uses of the shell including advanced source control operations, execution of long-running jobs, remote logins, and system administration of RStudio.

The **Terminal**pane is unlike most of the other features found in RStudio in that it's capabilities are platform specific. In general, these differences can be categorized as either Windows capabilities or other (Mac, Linux, RStudio Server).

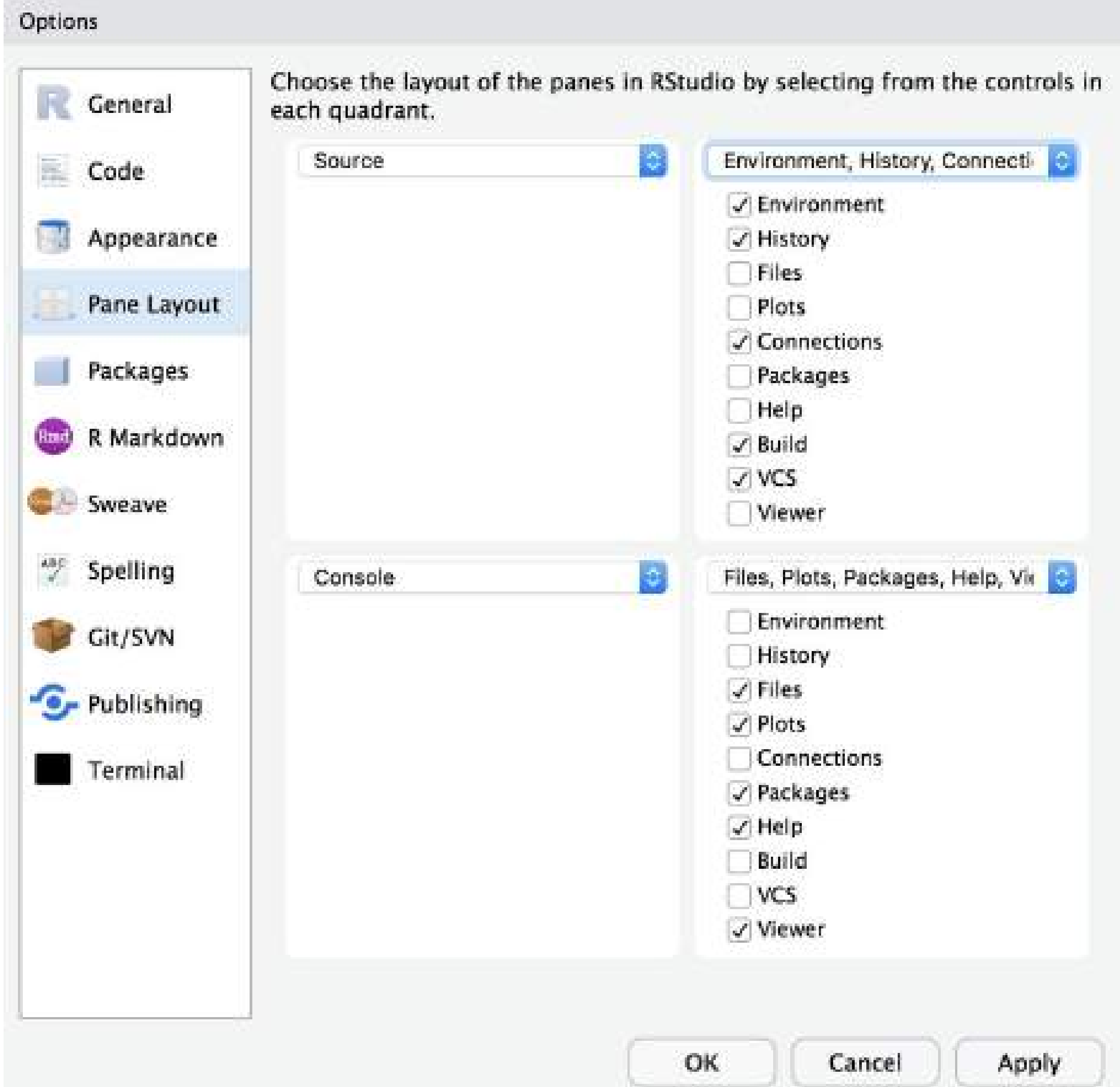
Customizing the Interface

If you don't like the default RStudio interface, you can customize the appearance. To do so, go to **Tool | Options** (**RStudio | Preferences**on a Mac).

The dialog seen in the screenshot below will be displayed.



The **Pane Layout** tab is used to change the locations of console, source editor, and tab panes, and set which tabs are included in each pane.



Menu Options

There are also a multitude of options that can be accessed from the RStudio menu items as well. Covering these items in depth is beyond the scope of this book, but in general here are some of the more useful functions that can be accessed through the menus.

1. Create new files and projects
2. Import datasets
3. Hide, show, and zoom in and out of panes
4. Work with plots (save, zoom, clear)

5. Set the working directory
6. Save and load workspace
7. Start a new session
8. Debugging tools
9. Profiling tools
10. Install packages
11. Access help system

You'll learn how to use various components of the RStudio interface as we move through the exercises in the book.

Installing RStudio

If you haven't already done so, now is a good time to download and install RStudio. There are a number of versions of RStudio, including a free open source version which will be sufficient for this book. Versions are also available for various operating systems including Windows, Mac, and Linux.

1. Go to https://www.rstudio.com/products/rstudio/download/ find RStudio for Desktop, the Open Source License version, and follow in the instructions to download and install the software.

In the next section we'll explore the basic programming constructs of the R language including the creation and assigning of data to variables, as well as the data types and objects that can be assigned to variables.

Installing the Exercise Data

This is intended as a hands-on exercise book and is designed to give you as much handson coding experience with R as possible. Many of the exercises in this book require that you load data from a file-based data source such as a CSV file. These files will need to be installed on your computer before continuing with the exercises in this chapter as well as the rest of the book. Please follow the instructions below to download and install the exercise data.

1. In a web browser go to https://www.dropbox.com/s/5p7j7nl8hgijsnx/ IntroR.zip?dl=0.
2. This will download a file called IntroR.zip.

3. The exercise data can be unzipped to any location on your computer. After unzipping the IntroR.zip file you will have a folder structure that includes IntroR

as the top-most folder with sub-folders called Data and Solutions. The Data folder contains the data that will be used in the exercises in the book, while the Solutions folder contains solution files for the R script that you will write.

RStudio can be used on Windows, Mac, or Linux so rather than specifying a specific folder to place the data I will leave the installation location up to you. Just remember where you unzip the data because you'll need to reference the location when you set the working directory.

4. For reference purposes I have installed the data to the desktop of my Mac computer under IntroR\Data. You will see this location referenced at various locations throughout the book. However, keep in mind that you can install the data anywhere.

Exercise 1: Creating variables and assigning data

In the R programming language, like other languages, variables are given a name and assigned data. Each variable has a name that represents its area in memory. In R, variables are case sensitive so use care in naming your variable and referring to them later in your code.

There are two ways that variables can be assigned in R. In the first code example below, a variable named x is created. The use of a less than sign immediately followed by a dash then precedes the variable name. This is the operator used to assign data to a variable in R. On the right-hand side of this operator is the value being assign to the variable. In this case, the value 10 has been assigned to the variable x. To print the value of a variable in R you can simple type the variable name and then click the **Enter** key on your keyboard.

```
x <- 10
x
[1] 10
```

The other way of creating and assigning data to a variable is to use the equal sign. In the second code example we create a variable called y and assign the value 10 to the variable. This second method of creating and assigning data to a variable is probably more familiar to you if you've used other languages like Python or JavaScript.

```
y = 10
y
[1] 10
```

In the R programming language, like other languages, variables are given a name and assigned data. Each variable is a named area in the computer's memory. In R, variables are also case sensitive so use care in naming your variables and referring to them later in your code. In this exercise you'll learn how to create variables in R and assign data. 1. Open RStudio and find the **Console** window. It should be on the left-hand

side of your screen at the bottom.

2. The first thing you'll need to do is set the working directory for the RStudio session. The working directory for all chapters in this book will be the location where you installed the exercise data. Please refer back to the section *Installing Exercise Data* for exercise data installation instructions if you haven't already completed this step.

The working directory can be set by typing the code you see below into the **Console** pane or by going to **Session | Set Working Directory | Choose Directory** from the RStudio menu. You will need to specify the location of the IntroR\Data folder where you installed

```
setwd(<installation directory for exercise data>)
```

3. As I mentioned in the introduction to this exercise, there are two ways to create and assign data to variables in R. We'll examine both in this section. First, create a variable called x and assign the value 10 as seen below. Notice the use of the less than sign (<) followed immediately by a dash (-). This operator can be used to assign data to a variable. The variable name is on the left-hand side of the operator, and the data we're assigning to the variable is on the right-hand side of the operator.

Note: The screenshot below displays a working directory of ~/Desktop/IntroR/Data/ which may or may not be your working directory. This is simply the working directory that I've defined for my RStudio session on a Mac computer. This will depend entirely on where you installed the exercise data for the book and the working directory you have set for your RStudio session.

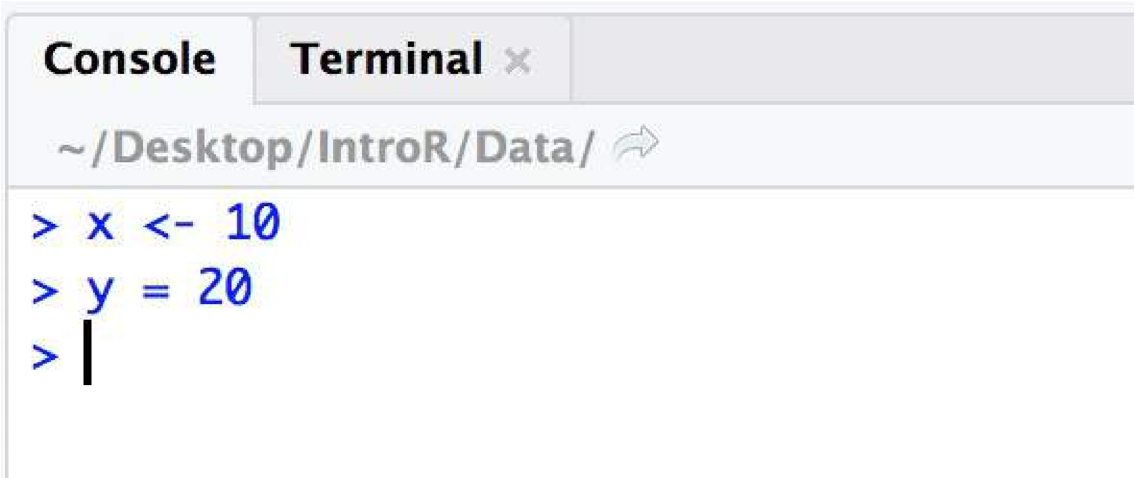
Console Terminal ×

~/Desktop/IntroR/Data/

```
> x <- 10
>
```

4. The second way of creating a variable is to use the equal sign. Create

a second variable using this method as seen in the screenshot below. Assign the value as y = 20. I will use the equal sign throughout the book in future exercises since it is used in other programming languages and is easier to understand and type. However, you are free to use either operator.



5. Finally, create a third variable called z and assign it the value of x + y. The variables x, y, and z have all been assigned numeric data. Variables in R can be assigned other types of data as well including characters (also known as strings), Booleans, and a number of data objects including vectors, factors, lists, matrices, data frames, and others.

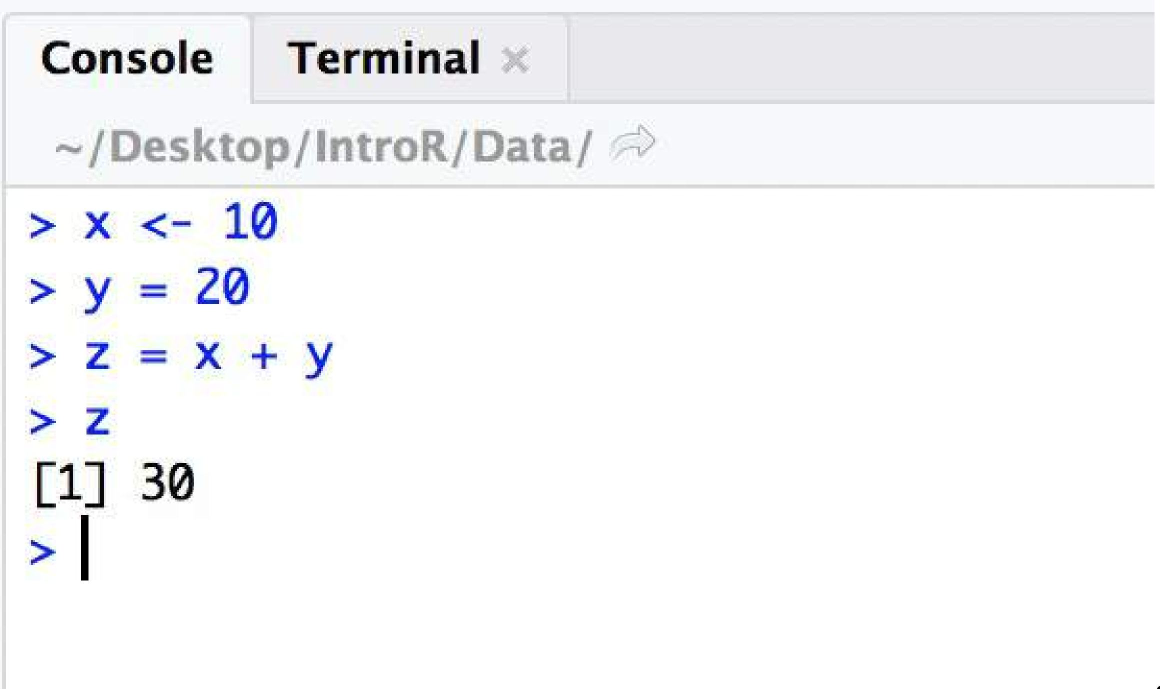
Console Terminal

~/Desktop/IntroR/Data/

```
> x <- 10
> y = 20
> z = x + y
>
```

6. The three variables that you've created (x, y, and z) are all numeric data types. This should be self-explanatory, but any number, including integers, floating point, and complex numbers are inherently defined as numeric data types. However, if you surround a number with quotes it will be interpreted by R as a character data type.

7. You can view the value of any variable simply by typing the variable name as seen in the screenshot below. Do that now to see how it works. Typing the name of a variable and clicking the **Enter\Return** key will implicitly call the print() function.

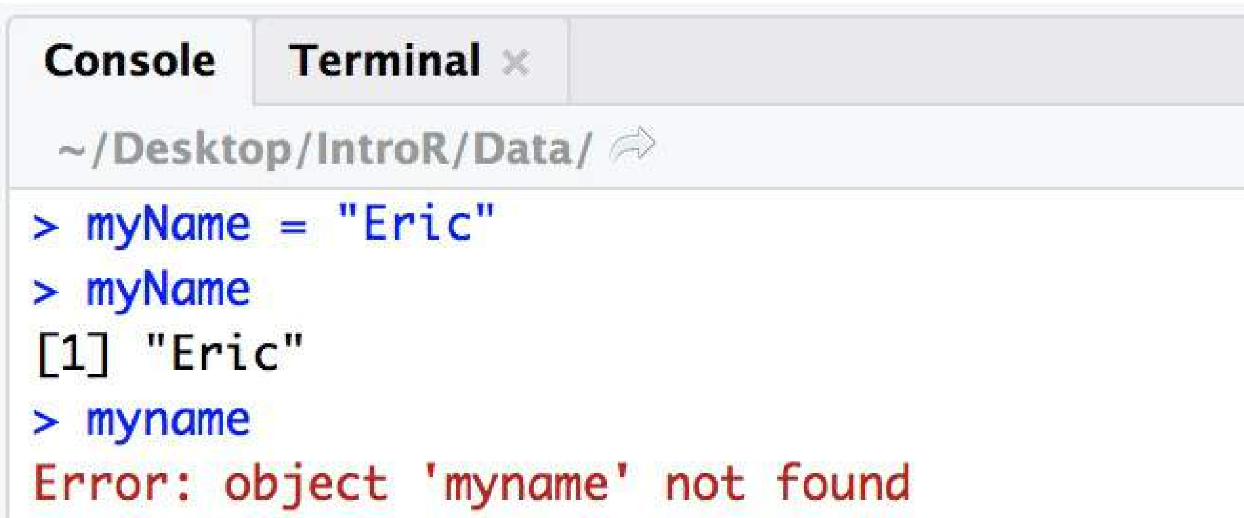


8.
The same thing can be accomplished using the print() function as seen below.

```
> print(z)
[1] 30
```

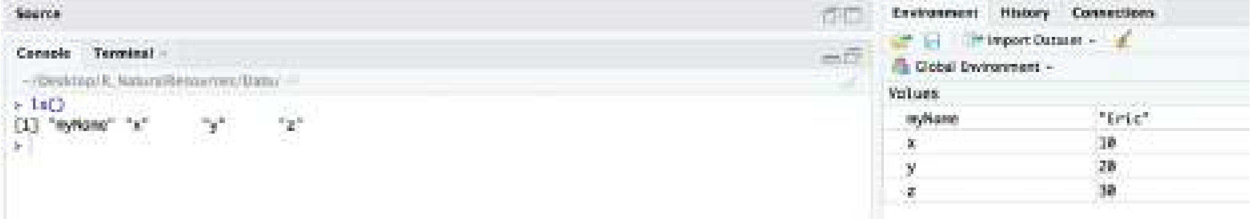
9. Variables in R are case sensitive. To illustrate this, create a new variable called myName and assign it the value of your name as I have done in the screenshot below. In this case, since we've enclosed the value with quotes, R will assign it as a character (string) data type. Any sequence of characters, whether they be letters, numbers, or special characters, will be defined as a character data type if surrounded by quotes.

Notice that when I type the name of the variable (with the correct case) it will report the value associated with the variable, but when I type myname (all lowercase) it reports an error. Even though the name is the same the casing is different, so you must always refer to your variable names with the same case that they were created.



10. To see a list of all variables in your current workspace you can type the

ls() function. Do that now to see a list of all the variables you have created in this session. Each variable and its current value is also displayed in the **Environment** pane on the right-hand side of RStudio.



11. There are many data types that can be assigned to variables. In this brief exercise we assigned both character (string) and numeric data to variables. As we dive further into the book we'll examine additional data types that can be assigned to variables in R. The syntax will remain the same though no matter what type of data is being assigned to a variable.

12. You can check your work against the solution file Chapter1_1.R.

Exercise 2: Using vectors and factors

In R, a vector is a sequence of data elements that have the same data type. Vectors are used primarily as container style variables used to hold multiple values that can then be manipulated or extracted as needed. The key though is

that all the values must be of the same type. For example, all the values must be numeric, character, or Boolean. You can't include any sort of combination of data types.

To create a vector in R you call the c() function and pass in a list of values of the same type. After creating a vector there are a number of ways that you can examine, manipulate, and extract data. In this exercise you'll learn the basics of working with vectors.

1. Open RStudio and find the **Console** pane. It should be on the left-hand side of your screen at the bottom.

2. In the R **Console** pane create a new vector as seen in the code example below. The c() function is used to create the vector object. This vector is composed of character data types. Remember that all values in the vector must be of the same data type.

layers <- c('Parcels', 'Streets', 'Railroads', 'Streams', 'Buildings')
3. Get the length of the vector using the length() function. This should return a value of 5.
length(layers) [1] 5

4. You can retrieve individual items from a vector by passing in an index number. Retrieve the Railroads value by passing in an index number of 3, which corresponds to the positional order of this value. R is a 1 based language so the first item in the list occupies position 1.

layers[3] [1] "Railroads"
5. You can extract a contiguous sequence of values by passing in two index numbers as seen below.
layers[3:5]
[1] "Railroads" "Streams" "Buildings"
6. Values can be removed from a vector by passing in a negative integer as seen below. This will remove Streams
from the vector.

layers
[1] "Parcels" "Streets" "Railroads" "Streams" "Buildings" layers[-4]
[1] "Parcels" "Streets" "Railroads" "Buildings"

7. Create a second vector containing numbers as seen below.
layerIds <- c(1,2,3,4)

8. In this next step we're going to combine the layers and layerIds vectors into a single vector. You'll recall that all the items in a vector must be of the same data type. In a case like this where one vector contains characters and the other numbers, R will automatically convert the numbers to characters. Enter the following code to see this in action.

```
layerIds <- c(1,2,3,4)
combinedVector <- c(layers, layerIds)
combinedVector
[1] “Parcels” “Streets” “Railroads” “Streams” “Buildings” [6] “1” “2” “3” “4”
```

9. Now let's create two new sets of vectors to see how vector arithmetic works. Add the following lines of code.
```
x <- c(10,20,30,40,50) y <- c(100,200,300,400,500)
```
10. Now add the values of the vectors.
```
x + y
[1] 110 220 330 440 550
```
11. Subtract the values.
```
y - x
[1] 90 180 270 360 450
```
12. Multiply the values.

```
10 * x
[1] 100 200 300 400 500
20 * y
[1] 2000 4000 6000 8000 10000
```

13. You can also use the built in R function against the values of a vector. Enter the follow lines of codes to see how the built-in functions work.
```
sum(x) [1] 150

mean(y)
[1] 300
median(y) [1] 300

max(y) [1] 500 min(x) [1] 10
```

14. A Factor is basically a vector but with categories, so it will look familiar to you. Go ahead and clear the R **Console** by selecting the **Edit** menu item and then **Clear Console** in RStudio.

15. Add the following code block. Note that you can easily use line continuation in R simply by selecting the Enter (Return) key on your keyboard. It will automatically add the "+" at the beginning of the line indicating that it is simply a continuation of the last line.

land.type <- factor(c("Residential", "Commercial", "Agricultural", "Commercial", "Commercial", "Residential"), levels=c("Residential", "Commercial"))

table(land.type) land.type
Residential Commercial 2 3

16. Now let's talk about ordering of factors. There may be times when you want to order the output of the factor. For example, you may want to order the results by month. Enter the following code:

mons <- c("March", "April", "January", "November", "January", + "September", "October", "September", "November", "August", + "January", "November", "November", "February", "May", "August", + "July", "December", "August", "August", "September", "November", + "February", "April")

mons <- factor(mons)
table(mons) mons

April August December February January July 2 4 1 2 3 1
March May November October September 1 1 5 1 3

17. The output is less than desirable in this case. It would be preferable to have the months listed in the order that they occur during the year. Creating an ordered factor resolves this issue. Add the following code to see how this works.

mons <- factor(mons, levels=c('January', 'February', 'March', + 'April', 'May', 'June', 'July', 'August', 'September', + 'October', 'November','December'), ordered=TRUE)

table(mons)
mons
January February March April May June

3 2 1 2 1 0 July August September October November December
1 4 3 1 5 1

Creating an ordered factor resolves this issue. In the next exercise you'll learn how to use lists, which are similar in many ways to vectors in that they are a container style object, but as you'll see they differ in an important way as well. You can check your work against the solution file Chapter1_2.R.

Exercise 3: Using lists

A list is an ordered collection of elements, in many ways very similar to vectors. However, there are some important differences between a list and a vector. With lists you can include any combination of data types. This differs from other data structures like vectors, matrices, and factors which must contain the same data type. Lists are highly versatile and useful data types. A list in R acts as a container style object in that it can hold many values that you store temporarily and pull out as needed.

1. Clear the R
Console by selecting the **Edit** menu item and then **Clear Console** in RStudio.

2. Lists can be created through the use of the list() function. It's also common to call a function that returns a list variable as well, but for the sake of simplicity in this exercise we'll use the list() function to create the list.

Each value that you intend to place inside the list should be separated by a comma. The values placed into the list can be of any type, which differs from vectors that must all be of the same type. Add the code you see below in the **Console** pane.

my.list <- list("Streets", 2000, "Parcels", 5000, TRUE, FALSE)
In this example a list called my.list
has been created with a number of character, numeric, and Boolean values.

3. Because lists are container style objects you will need to pull values out of a

list at various times. This is done by passing an index number inside square brackets, with the index number one referring to the first value in the list, and each successive value occupying the next index number in order. However, accessing items in a list can be a little confusing as you'll see. Add the following code and then we'll discuss.

```
my.list[2] [[1]]
[1] 2000
```

The index number 2 is a reference to the second value in the my.list object, which in this case is the number 2000. However, when you pass an index number inside a **single pair of square braces** it actually returns another list object, this time with a single value. In this case, 2000 is the only value in the list, but it is a list object rather than a number.

4. Now add the code you see below to see how to pull out the actual value from the list rather than returning another list with a single value.

```
my.list[[2]]
```

In this case we pass a value of 2 inside a pair of square braces. Using two square braces on either side of the index number will pull the actual value out of the list rather than returning a new list with a single value. In this case, the value 2000 is returned as a numeric value. This can be a little confusing the first few times you see and use this, but lists are a commonly used data type in R so you'll want to make sure you understand this concept.

5. There may be times when you want to pull multiple values from a list rather than just a single value. This is called list slicing and can be accomplished using syntax you see below. In this case we pass in two index numbers that indicate the starting and ending position of the values that should be retrieved. Try this on your own.

```
new.list <- my.list[c(1,2)] new.list
[[1]]
[1] “Streets”

[[2]]
[1] 2000
```

6. This returned a new list object stored in the variable new.list. Using basic list indexing you can then pull a value out of this list.

new.list[[2]] [1] 2000

7. You can get the number of items in a list by calling the length() function. This will return the number of values in the list, not including any nested lists. Calling the length() function in this exercise on the my.list variable should produce a result of 6.

length(my.list)

8. Finally, there may be times when you are uncertain if a variable is stored as a vector or a list. You can use the is.list() function, which will return a TRUE or FALSE value that indicates whether the variable is a list object.

is.list(my.list) [1] TRUE
9. You can check your work against the solution file Chapter1_3.R.

Exercise 4: Using data classes

In this exercise we'll take a look at matrices and data frames. A matrix in R is a structure very similar to a table in that it has columns and rows. This type of structure is commonly used in statistical operations. A matrix is created using the matrix() function. The number of columns and rows can be passed in as arguments to the function to define the attributes and data values of the matrix. A matrix might be created from the values found in the attribute table of a feature class. However, keep in mind that all the values in the matrix must of the same data type.

Data frames in R are very similar to tables in that they have columns and rows. This makes them very similar to matrix objects as well. In statistics, a dataset will often contain multiple variables. For example, if you are analyzing real estate sales for an area there will be many factors including income, job growth, immigration, and others.

These individual variables are stored as the columns in a data frame. Data frames are most commonly created by loading an external file, database table, or URL containing tabular information using one of the many functions provided by R for importing a dataset. You can also manually enter the values. When manually entering the data the R console will display a spreadsheet style interface that you

can use to define the column names as well as the row values. R includes many built-in datasets that you can use for learning purposes and these are stored as data frames.

1. Open RStudio and find the
Console pane. It should be on the bottom, lefthand side of your screen.

2. Let's start with matrices. In the R **Console** create a new matrix as seen in the code example below. The c() function is used to define the data for the object. This matrix is composed of numeric data types. Remember that all values in the matrix must be of the same data type.

matrx <- matrix(c(2,4,3,1,5,7), nrow=2, ncol=3, byrow=TRUE) matrx

[,1] [,2] [,3] [1,] 2 4 3
[2,] 1 5 7

3. You can name the columns in a matrix. Add the code you see below to name your columns.

colnames(matrx) <- c("POP2000", "POP2005", "POP2010") POP2000
POP2005 POP2010
[1,] 2 4 3
[2,] 1 5 7

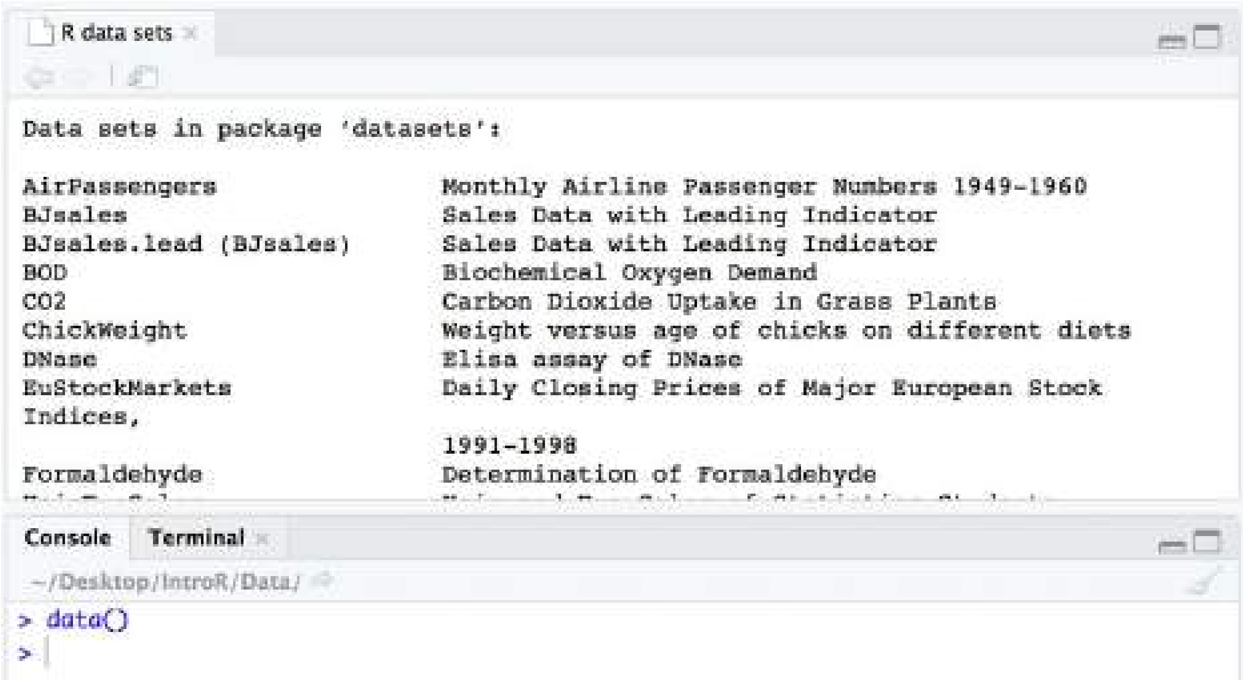
4. Now let's retrieve a value from the matrix with the code you see below. The format is
matrix(row, column).
matrx[2,3]
POP2010
7
5. You can also extract an entire row using the code you see below. Here we just provide a row value but no column.
matrx[2,]
POP2000 POP2005 POP2010 1 5 7
6. Or you can extract an entire column using the format you see below.
matrx[,3] [1] 3 7
7. You can also extract multiple columns at a time.
matrx[,c(1,3)]

POP2000 POP2010
[1,] 2 3
[2,] 1 7

8. You can also access columns or rows by name if you have named them.
matrx[, “POP2005”] [1] 4 5
9. You can use the
colSums(), colMeans() or rowSums()
functions against the data as well.

colSums(matrx)
POP2000 POP2005 POP2010
3 8 11
> colMeans(matrx)
POP2000 POP2005 POP2010
1.5 4.0 5.5

10. Now we’ll turn our attention to Data Frames. Clear the R console and execute the data() function as seen below. This displays a list of all the sample datasets that are part of R. You can use any of these datasets.



11. For this exercise we’ll use the USArrests data frame. Add the code you see below to display the contents of the USArrests data frame.

```
> data()
> data("USArrests")
> USArrests
            Murder Assault UrbanPop Rape
Alabama       13.2     236       58 21.2
Alaska        10.0     263       48 44.5
Arizona        8.1     294       80 31.0
Arkansas       8.8     190       50 19.5
California     9.0     276       91 40.6
Colorado       7.9     204       78 38.7
Connecticut    3.3     110       77 11.1
Delaware       5.9     238       72 15.8
Florida       15.4     335       80 31.9
Georgia       17.4     211       60 25.8
Hawaii         5.3      46       83 20.2
Idaho          2.6     120       54 14.2
Illinois      10.4     249       83 24.0
Indiana        7.2     113       65 21.0
```

12. Next, we'll pull out the data for all rows from the Assault column.

USArrests$Assault
[1] 236 263 294 190 276 204 110 238 335 211 46 120 249 113 56
115
[17] 109 249 83 300 149 255 72 259 178 109 102 252 57 159 285
254
[33] 337 45 120 151 159 106 174 279 86 188 201 120 48 156 145
81
[49] 53 161

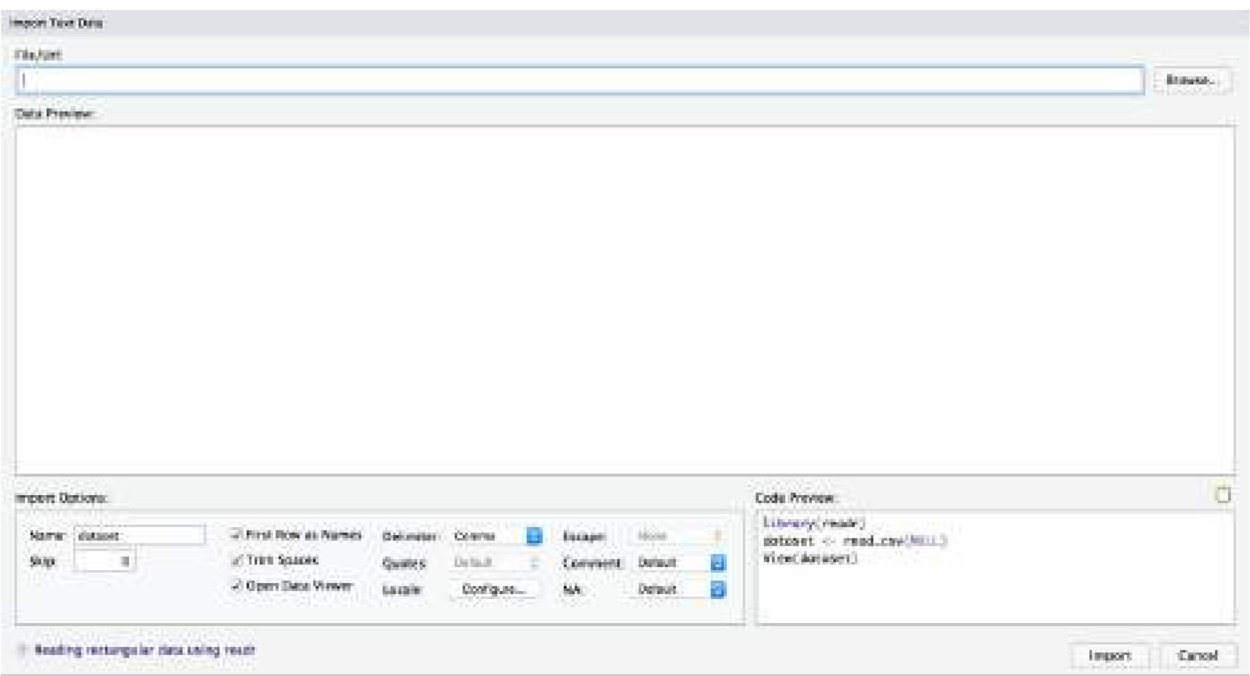
13. A value from a specific row, column combination can be extracted using the

code seen below where the row is specified as the first offset and the column is the second. This particular code extracts the assault value for Wyoming.

USArrests[50,2] [1] 161
14. If you leave off the column it will return all columns for that row.
USArrests[50,]
Murder Assault UrbanPop Rape Wyoming 6.8 161 60 15.6

The sample datasets included with R are good for learning purposes, but of limited usefulness beyond that. You’re going to want to load datasets that are relevant to your line of work, and many of these datasets have a tabular structure that is conducive to the data frame object. Most of these datasets will need to be loaded from an external source that may be found in delimited text files, database tables, web services, and others. You’ll learn how to load these external datasets using R code in a later chapter of the book, but as you’ll see in this next exercise you can also use the RStudio interface to load them as well. 15. In RStudio go to the **File** menu and select **Import Dataset | From Text**

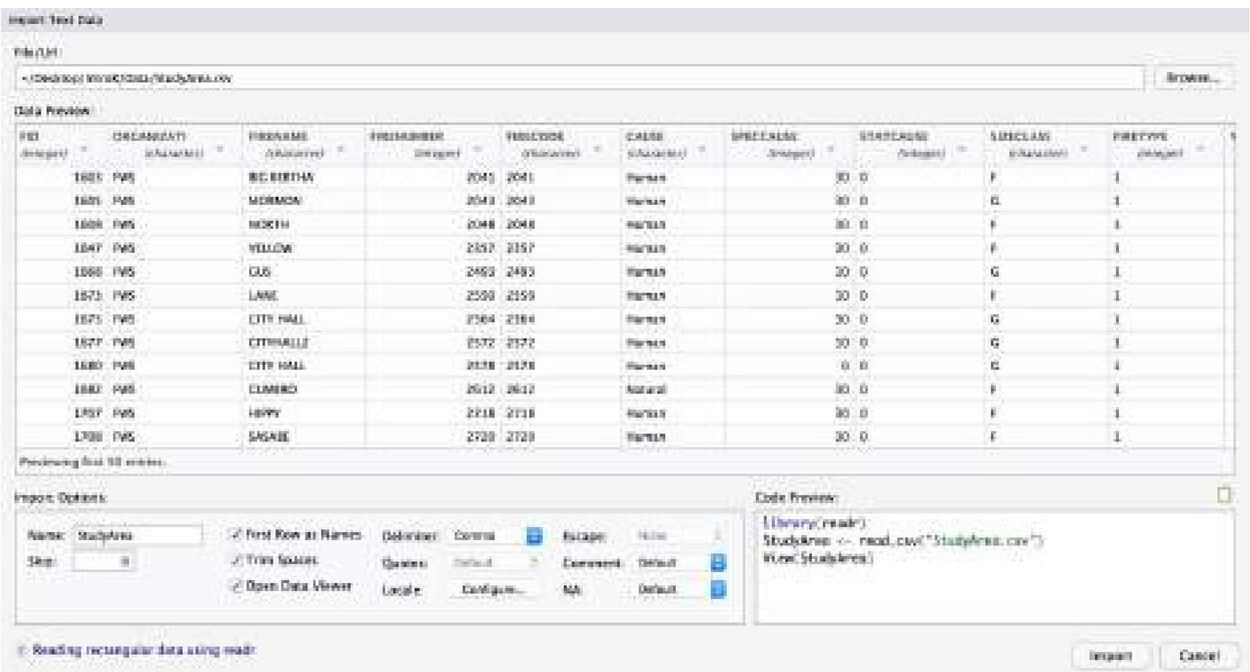
(readr) . This will display the dialog seen in the screenshot below. We’ll discuss the readr package in much more detail in a future chapter, but this package is used to efficiently read external data into a data frame.



16. Use the **Browse** button to browse to the StudyArea.csv file found in the Data

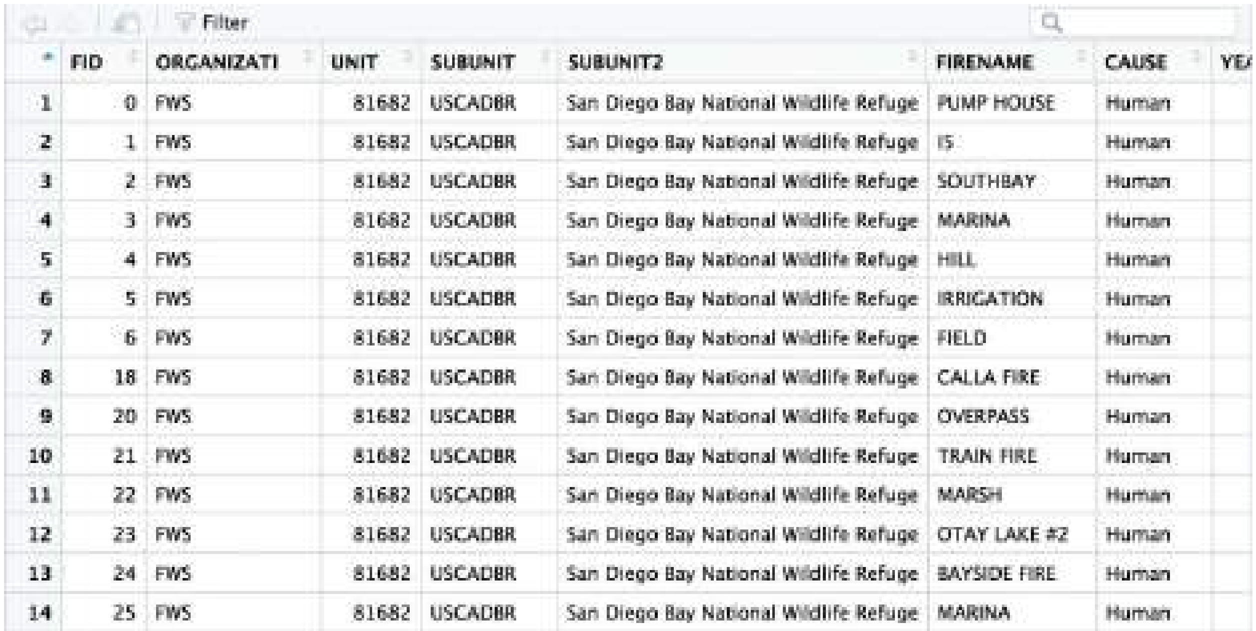
folder where you installed the exercise data for this book. The StudyArea.csv file is a comma separated list of wildfires from 1980-2016 for the Western United States.

The data will be loaded into a preview window as seen below. There are a number of import options along with the code that will be executed. You can leave the default values in this case.



17. Click **Import** from this **Import Test Data** dialog. This will load the data

into a data frame (technically called a Tibble in tidyverse) called StudyArea. It will also use the View() function to display the results in a tabular view displayed in the screenshot below.



	FID	ORGANIZATI	UNIT	SUBUNIT	SUBUNIT2	FIRENAME	CAUSE	YEA
1	0	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	PUMP HOUSE	Human	
2	1	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	I5	Human	
3	2	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	SOUTHBAY	Human	
4	3	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	MARINA	Human	
5	4	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	HILL	Human	
6	5	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	IRRIGATION	Human	
7	6	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	FIELD	Human	
8	18	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	CALLA FIRE	Human	
9	20	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	OVERPASS	Human	
10	21	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	TRAIN FIRE	Human	
11	22	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	MARSH	Human	
12	23	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	OTAY LAKE #2	Human	
13	24	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	BAYSIDE FIRE	Human	
14	25	FWS	81682	USCADBR	San Diego Bay National Wildlife Refuge	MARINA	Human	

18. Messages, warnings, and errors from the import will be displayed in the

Console window. You can ignore these messages for now. We'll discuss them in more detail in a later chapter.

```
> StudyArea <- read_csv("StudyArea.csv")
Parsed with column specification:
cols(
  .default = col_character(),
  FID = col_integer(),
  SPECCAUSE = col_integer(),
  STATCAUSE = col_integer(),
  FIRETYPE = col_integer(),
  YEAR_ = col_integer(),
  STATE_FIPS = col_integer(),
  DLATITUDE = col_double(),
  DLONGITUDE = col_double(),
  TOTALACRES = col_double(),
  TRPGENCAUS = col_integer(),
  TRPSPECCAU = col_integer()
)
See spec(...) for full column specifications.
```

This StudyArea data frame can then be used for data exploration and visualization, which we'll cover in future chapters.

19. You can check your work against the solution file Chapter1_4.R.

Exercise 5: Looping statements

Looping statements aren't used as much in R as they are in other languages because R has built in support for vectorization. Vectorization is a built-in structure that automatically loops through a data structure without the need to write looping code. However, there may be times when you need to write looping code to accomplish a specific task that isn't handled by vectorization so you need to understand the syntax of looping statements in R. We'll take a look at a simple block of code that loops through the rows in a data frame.

For loops are used when you know exactly how many times to repeat a block of code. This includes the use of data frame objects that have a specific number of rows. For loops are typically used with vector and data frame structures.

1. For this brief exercise we'll use the StudyArea data frame that you imported from an external file in the last exercise. You will also learn how to create an R script and learn how to execute the script. A script is simply a series of commands that are run as a group rather than entering and running your code one line at a time from the **Console** window.

2. Create a new R script by going to **File | New File | R Script** from the RStudio interface.

3. Save the file with a name of Chapter1_5.R. You can place the script file wherever you'd like, but it is recommended that you save it to your folder where your exercise data is loaded.

4. Add the following lines of code to the Chapter1_5.R script.

```
for (fire in 1:nrow(StudyArea)) { print(StudyArea[fire, "TOTALACRES"])
}
```

5. Run the code by selecting **Code | Run Region | Run All** from the RStudio menu or by clicking the **Source** button on the script tab.

This will produce a stream of data that looks similar to what you see below. You will want to stop the execution of this script after it begins displaying data because of the amount of data and time it will take to print out all the information. The for loop syntax assigns each row from the StudyArea data frame to a variable called fire. The total number of acres burned for each fire is then printed.

```
# A tibble: 1 x 1 TOTALACRES

<dbl>
1 0.100
# A tibble: 1 x 1
```

TOTALACRES
<dbl>
1 3.
A tibble: 1 x 1

TOTALACRES
<dbl>
1 0.500
A tibble: 1 x 1

TOTALACRES
<dbl>
1 0.100
A tibble: 1 x 1

TOTALACRES
<dbl>

As I mentioned earlier, you won't often need to use for loops in R because of the built-in support for vectorization, but sooner or later you'll run into a situation where you need to create these looping structures.

6. You can check your work against the solution file Chapter1_5.R.

Exercise 6: Decision support statements – if | else

Decision support statements enable you to write code that branches based upon specific conditions. The basic if | elsestatement in R is used for decision support. Basically, ifstatements are used to branch code based on a test expression. If the test expression evaluates to TRUE, then a block of code is executed. If the test evaluates to FALSE then the processing skips down to the first else if statement or an elsestatement if you don't include any else if statements.

Each if | else if | elsestatement has an associated code block that will execute when the statement evaluates to TRUE. Code blocks are denoted in R using curly braces as seen in the code example below.

You can include zero or more else ifstatements depending on what you're

attempting to accomplish in your code. If no statements evaluate to TRUE, processing will execute the code block associated with the else statement.
1. In this exercise we'll build on the looping exercise by adding in an if |

else if | else block that displays the fire names according to size. 2. Create a new R script by going to **File | New File | R Script** from the RStudio interface.

3. Save the file with a name of Chapter1_6.R. You can place the script file wherever you'd like, but it is recommended that you save it to your folder where your exercise data is loaded.

4. Copy and paste the for loop you created in the last exercise and saved to the Chapter1_5.R file into your new Chapter1_6.R
file.

```
for (fire in 1:nrow(StudyArea)) { print(StudyArea[fire, "TOTALACRES"])
}
```

5. Add the if | else if block you see below. This script loops through all the rows in the StudyArea data frame and prints out messages that indicate when a fire has burned more than the specified number of acres for each category.

```
for (fire in 1:nrow(StudyArea)) {
if(StudyArea[fire, "TOTALACRES"] > 100000) {
print(paste("100K Fire: ", StudyArea[fire, "FIRENAME"], sep = ""))
}
else if (StudyArea[fire, "TOTALACRES"] > 75000) {
print(paste("75K Fire: ", StudyArea[fire, "FIRENAME"], sep = ""))
} else if (StudyArea[fire, "TOTALACRES"] > 50000) {
print(paste("50K Fire: ", StudyArea[fire, "FIRENAME"], sep =

""))
}
}
```

6. Run the code by selecting **Code | Run Region | Run All** from the RStudio menu or by clicking the **Source** button on the script tab. The script should start producing output in the **Console** pane similar to what you see below.

```
[1] "50K Fire: PIRU"
```

[1] “100K Fire: CEDAR”
[1] “50K Fire: MINE”
[1] “100K Fire: 24 COMMAND”
[1] “50K Fire: RANCH”
[1] “75K Fire: HARRIS”
[1] “50K Fire: SUNNYSIDE TURN OFF” [1] “100K Fire: Range 12”

7. You can optionally add an else block at the end that will print a message for any fire that isn’t greater than 50,000 acres. Most of the fires in this dataset are less than 50,000 so you’ll see a lot of messages that indicate this if you add the else block below.

```
for (fire in 1:nrow(StudyArea)) {
if(StudyArea[fire, “TOTALACRES”] > 100000) {
print(paste(“100K Fire: “, StudyArea[fire, “FIRENAME”], sep = “”)) }
else if (StudyArea[fire, “TOTALACRES”] > 75000) {
print(paste(“75K Fire: “, StudyArea[fire, “FIRENAME”], sep = “”)) }
else if (StudyArea[fire, “TOTALACRES”] > 50000) {
print(paste(“50K Fire: “, StudyArea[fire, “FIRENAME”], sep = “”)) }
else {
print(“Not a MEGAFIRE”)
}
}
```

8. You can check your work against the solution file Chapter1_6.R.

Exercise 7: Using functions

Functions are a group of statements that execute as a group and are action-oriented structures in that they accomplish some sort of task. Input variables can be passed into functions through what are known as parameters. Another name for parameters is arguments. These parameters become variables inside the function to which they are passed.

R packages include many pre-built functions that you can use to accomplish specific tasks, but you can also build your own functions. Functions take the form seen in the screenshot below.

```
myfunction <- function(arg1, arg2, ... ){
statements
return(object)
}
```

Functions are assigned a name, can take zero or more arguments, each separated by a comma, have a body of statements that execute as a group, and can return a value. The body of a function is always enclosed by curly braces. This is where the work of the function is accomplished. Any variables defined inside the function or passed as arguments to the function become local variables that are only accessible from inside the function. The return keyword is used to return a value to the code that initially called the function.

The way you call a function can differ a little. The basic form of calling a function is to reference the name of the function followed by any arguments inside parentheses just after the name of the function. When passing arguments to the function using this default syntax, you simply pass the value for the parameter, and it is assumed that you are passing them in the order that they were defined. In this case the order that you pass in the arguments is very important. The order must match the order that was used to define the function. This is illustrated in the code example below.

myfunction(2, 4)
If the function returns a value, then you will need to assign a variable name to the function call as seen in the code example below that creates a variable called
z
.
z = myfunction(2, 4)

Finally, while you don't have to specify the name of the argument you can do so if you'd like. In this case you simply pass in the name of the argument followed by an equal sign and then the value being passed for that argument. The code example below illustrates this optional way of calling a function.

myfunction(arg1=2, arg2 = 4)
In this exercise you'll learn how to call some of the built-in R functions.

1. R includes a number of built in functions for generating summary statistics for a dataset. In this exercise we'll call some of the functions on the StudyArea data frame that was created in *Exercise 4: Using Data Classes*. In the Console pane add the line of code you see below to call the mean() function. In this case, the TOTALACRES column from the StudyArea data frame will be passed as a parameter to the function. This function calculates the mean of a numeric dataset, which in this case will be 191.0917.

```
mean(StudyArea$TOTALACRES) [1] 191.0917
```

2. Repeat this same process with the min(), max(), and median() functions.

3. The YEAR_ field in the StudyArea data frame contains the year in which the fire occured. The substr() function can be used to extract a series of characters from a variable. Use the substr() function as seen below to extract out the last two digits of the year.

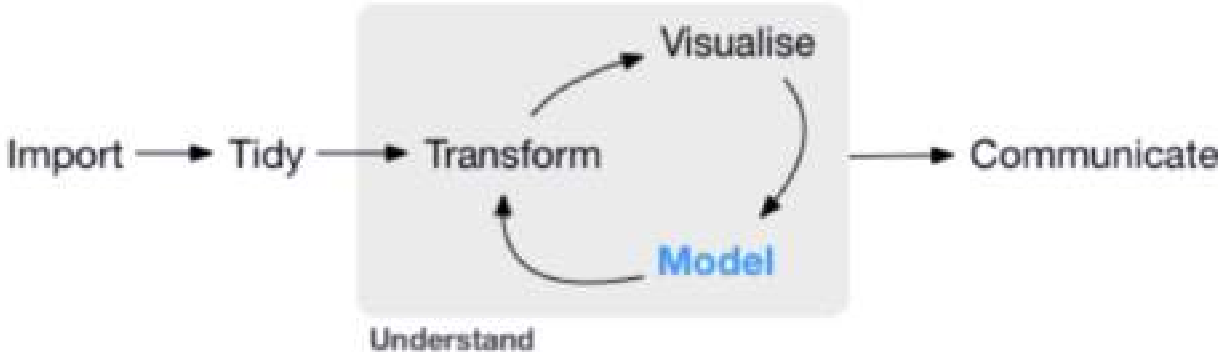
```
substr(StudyArea$YEAR_, 3, 4)
```

4. You've seen examples of a number of other built in R functions in previous exercises including print(), ls() rm(), and others. The base R package contains many functions that can be used to accomplish various tasks. There are thousands of other third-party R packages that you can use as well, and they all contain additional functions for performing specific tasks. You can also create your own functions, and we'll do that in a future chapter.

5. You can check your work against the solution file Chapter1_7.R.

Exercise 8: Introduction to tidyverse

While the base R package includes many useful functions and data structures that you can use to accomplish a wide variety of data science tasks, the third-party tidyverse package supports a comprehensive data science workflow as illustrated in the diagram below. The tidyverse ecosystem includes many sub-packages designed to address specific components of the workflow.



Ttidyverse is a coherent system of packages for importing, tidying, transforming, exploring, and visualizing data. The packages of the tidyverse ecosystem were mostly developed by Hadley Wickham, but they are now being expanded by several contributors. Tidyverse packages are intended to make statisticians and data scientists more productive by guiding them through workflows that facilitate communication, and result in reproducible work products. Fundamentally, the tidyverse is about the connections between the tools that make the workflow possible.
Let's briefly discuss the core packages that are part of tidyverse, and then we'll do a deeper dive into the specifics of the packages as we move through the book. We'll use these tools extensively throughout the book.

readr

The goal of readris to facilitate the import of file-based data into a structured data format. The readrpackage includes seven functions for importing file-based datasets including csv, tsv, delimited, fixed width, white space separated, and web log files.

Data is imported into a data structure called a tibble. Tibbles are the tidyverse implementation of a data frame. They are quite similar to data frames, but are basically a newer, more advanced version. However, there are some important differences between tibbles and data frames. Tibbles never convert data types of variables. They never change the names of variables or create row names. Tibblesalso have a refined print method that shows only the first 10 rows, and all columns that will fit on the screen. Tibblesalso print the column type along with the name. We'll refer to tibbles as data frames throughout the remainder of the book to keep things simple, but keep in mind that you're actually going to be working with tibbleobjects. In the next chapter you'll learn how to use the

read_csv() function to load csv files into a tibble object.

tidyr

Data tidying is a consistent way of organizing data in R, and can be facilitated through the tidyr package. There are three rules that we can follow to make a dataset tidy. First, each variable must have its own column. Second, each observation must have its own row, and finally, each value must have its own cell.

dplyr

The dplyr package is a very important part of tidyverse. It includes five key functions for transforming your data in various ways. These functions include filter(), arrange(), select(), mutate(), and summarize(). In addition, these functions all work very closely with the group_by()function. All five functions work in a very similar manner where the first argument is the data frame you're operating on, and the next N number of arguments are the variables to include. The result of calling all five functions is the creation of a new data frame that is a transformed version of the data frame passed to the function. We'll cover the specifics of each function in a later chapter.

ggplot2

The ggplot2package is a data visualization package for R, created by Hadley Wickham in 2005 and is an implementation of Leland Wilkinson's Grammar of Graphics.

Grammar of Graphics is a term used to express the idea of creating individual blocks that are combined into a graphical display. The building blocks used in ggplot2 to implement the Grammar of Graphics include data, aesthetic mapping, geometric objects, statistical transformations, scales, coordinate systems, position adjustments, and faceting.

Using ggplot2you can create many different kinds of charts and graphs including bar charts, box plots, violin plots, scatterplots, regression lines, and more. There are a number of advantages to using ggplot2versus other visualization techniques available in R. These advantages include a consistent style for defining the graphics, a high level of abstraction for specifying plots, flexibility, a built-in theming system for plot appearance, mature and complete graphics system, and access to many other ggplot2 users for support.

Other tidyverse packages

The tidyverse ecosystem includes a number of other supporting packages including stringr, purr, forcats, and others. In this book we'll focus primarily on the package already described, but to round out your knowledge of tidyverse you can reference tidyverse.org.

Conclusion

In this chapter you learned the basics of using the RStudio interface for data visualization and exploration as well as some of the basic capabilities of the R language. After learning how to create variables and assign data, you learned some of the basic R data types including characters, vectors, factors, lists, matrices, and data frames. You also learned about some of the basic programming constructs including looping, decision support statements, and functions. Finally, you received an overview of the tidyverse package. In the next chapter you'll learn some basic data exploration and visualization techniques before we dive into the specifics in future chapters.

Chapter 2

The Basics of Data Exploration and Visualization with R

Now that you've gotten your feet wet with the basics of R we're going to turn our attention to covering some of the fundamental concepts of data exploration and visualization using tidyverse. This chapter is going to be a gentle introduction to some of the topics that we're going to cover in much more exhaustive detail in coming chapters. For now, I just want you to get a sense of what is possible using various tools in the tidyverse package.

This chapter will teach you fundamental techniques for how to use the readr package to load external data from a CSV file into R, the dplyr package to massage and manipulate data, and ggplot2to visualize data. You'll also learn how to install and the tidyverse ecosystem of packages and load the packages into the RStudio environment.

As I mentioned previously, this chapter is intended as a gentle introduction to what is possible rather than a detailed inspection of the packages. Future chapters will go into extensive detail on these topics. For now, I just want you to get a sense of what is possible even if you don't completely understand the details.

In this chapter we'll cover the following topics:

- Installing and loading tidyverse
- Loading and examining a dataset
- Filtering a dataset
- Grouping and summarizing a dataset
- Plotting a dataset

Exercise 1: Installing and loading tidyverse

In *Chapter 1: Introduction to R* you learned the basics concepts of the tidyverse package. We'll be using various packages from the tidyverse ecosystem throughout this book including readr, dplyr, and ggplot2 among others. Tidyverse is a third-party package so you'll need to install the package using RStudio so that it can be used in the exercises in this book. In this exercise you'll

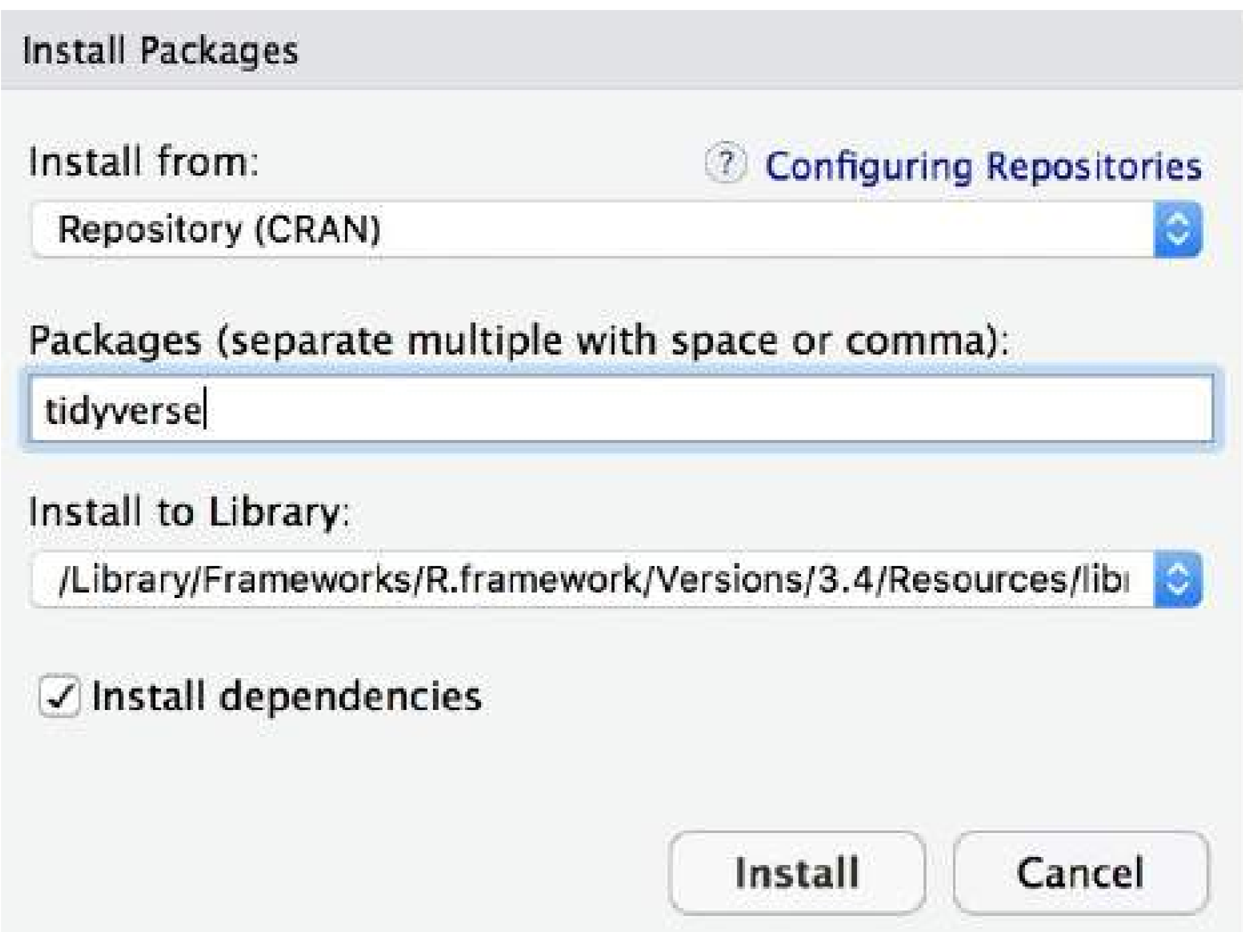
learn how to install tidyverse and load the package into your scripts.

1. Open RStudio.

2. The tidyverse package is really more an ecosystem of packages that can be used to carry out various data science tasks. When you install tidyverse it installs all of the packages that are part of tidyverse, many of which we discussed in the last chapter. Alternatively, you can install them individually as well. There are a couple ways that you can install packages in RStudio.

Locate the **Packages** pane in the lower right portion of the RStudio window. To install a new package using this pane, click the **Install** button shown in the screenshot below.

In the **Packages** textbox, type **tidyverse**. Alternatively, you can load the packages individually so instead of typing tidyverse you would type readr or ggplot2 or whatever package you want to install. We're going to use the readr, dplyr, and ggplot2 packages in this chapter and in many others so you can either install the entire tidyverse package, which includes the packages we'll use in this chapter plus a number of others or install them individually. Go ahead and do that now.



3. The other way of installing packages is to use the install.packages() function as seen below. This function should be types from the **Console** pane.

install.packages(<package>)
For example, if you wanted to install the dplyr package you would type:
install.packages("dplyr")

4. To use the functionality provided by a package it also needs to be loaded either into an individual script that will use the package, or it can also be loaded from the **Packages** pane. To load a package from the **Packages** pane, simply click the checkbox next to the package as seen in the screenshot below.

	Name	Description	Version
☐	RColorBrewer	ColorBrewer Palettes	1.1-2
☐	Rcpp	Seamless R and C++ Integration	0.12.16
☐	RCurl	General Network (HTTP/FTP/...) Client Interface for R	1.95-4.10
☑	readr	Read Rectangular Text Data	1.1.1
☐	readxl	Read Excel Files	1.0.0
☐	rematch	Match Regular Expressions with a Nicer 'API'	1.0.1
☐	reprex	Prepare Reproducible Example Code for Sharing	0.1.2
☐	reshape	Flexibly Reshape Data	0.8.7
☐	reshape2	Flexibly Reshape Data: A Reboot of the Reshape Package	1.4.3
☑	rgdal	Bindings for the 'Geospatial' Data Abstraction Library	1.3-1
☐	rgeos	Interface to Geometry Engine - Open Source ('GEOS')	0.3-26

5. You can also load a package from either a script or the **Console** pane by typing library(<package>). For example, to load the readr package you would type the following:

```
library(readr)
```

Exercise 2: Loading and examining a dataset

The tidyverse package is designed to work with data stored in an object called a Tibble. Tibbles are the tidyverse implementation of a data frame. They are quite similar to data frames, but are basically a newer, more advanced version.

There are some important differences between tibbles and data frames. Tibbles never convert the data types of variables. Also, they never change the names of variables or create row names. Tibblesalso have a refined print method that shows only the first 10 rows, and all columns that will fit on the screen. Tibbles also print the column type along with the name.
We'll refer to tibbles as data frames throughout the remainder of this chapter to keep things simple, but keep in mind that you're actually going to be working with tibble objects as opposed to the older data frame objects.

Getting data into a tibbleobject for manipulation, analysis, and visualization is normally accomplished through the use of one of the read functions found in the readr package. In this exercise you'll learn how to read the contents of a CSV file into R using the read_ csv() function found in the readr package.

1. Open R Studio.

2. In the **Packages** pane scroll down until you see the readr package and check the box just to the left as seen below as seen in the screenshot from the last exercise in this chapter. Note: If you don't see the readr package in the **Packages** pane it means that the package hasn't been installed. You'll need to go back to the last exercise and follow the instructions provided.

3. You will also need to set the working directory for the RStudio session. The easiest way to do this is to go to **Session | Set Working Directory | Choose Directory** and then navigate to the IntroR\Data folder where you installed the exercise data for this book.

4. The read_csv() function is going to be used to read the contents of a file called Crime_Data.csv. This file contains approximately 481,000 crime reports from Seattle, WA covering a span of approximately 10 years. If you have Microsoft Excel or some other spreadsheet type software take a few moments to examine the contents of this file.

For each crime offense this file includes date and time information, crime categories and description, police department information including sector, beat, and precinct, and neighborhood name.

5. Find the RStudio **Console** pane and add the code you see below. This will read the data stored in the Crime_Data.csv file into a data frame (actually a tibble as discussed in the introduction) called dfCrime.

```
dfCrime = read_csv("Crime_Data.csv", col_names = TRUE)
```

6. You'll see some messages indicating the column names and data types for each as seen below.

```
Parsed with column specification:
cols(
`Report Number` = col_double(),
`Occurred Date` = col_character(),
`Occurred Time` = col_integer(),
`Reported Date` = col_character(),
`Reported Time` = col_integer(),
`Crime Subcategory` = col_character(),
`Primary Offense Description` = col_character(),
Precinct = col_character(),
```

Sector = col_character(),
Beat = col_character(),
Neighborhood = col_character()
)

7. You can get a count of the number of records with the
nrow() function.
nrow(dfCrime) [1] 481376
8. The View() function can be used to view the data in a tabular format as seen in the screenshot below.
View(dfCrime)

	Report Number	Occurred Date	Occurred Time	Reported Date	Reported Time	Crime Subcategory	Primary Offense De
1	2.008e+13	12/13/1908	2114	12/13/2008	2114	DUI	DUI-LIQUOR
2	2.010e+13	06/15/1964	0	06/15/2010	1031	FAMILY OFFENSE-NONVIOLENT	CHILD-OTHER
3	2.012e+12	01/01/1973	0	01/25/2012	1048	SEX OFFENSE-OTHER	SEXOFF-OTHER
4	2.013e+13	06/01/1974	0	09/09/2013	1117	SEX OFFENSE-OTHER	SEXOFF-OTHER
5	2.016e+13	01/01/1975	0	08/11/2016	1054	SEX OFFENSE-OTHER	SEXOFF-OTHER
6	1.975e+12	12/16/1975	900	12/16/1975	1500	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-R

9. It will often be the case that you don't need all the columns in the data that you import. The dplyr package includes a select() function that can be used to limit the fields in the data frame. In the **Packages** pane, load the dplyr library. Again, if you don't see the dplyr library then it (or the entire tidyverse) will need to be installed.

10. In this case we'll limit the columns to the following: Reported Date,

Crime Subcategory , Primary Offense Description, Precinct, Sector, Beat, and Neighborhood. Add the code you see below to accomplish this.

dfCrime = select(dfCrime, 'Reported Date', 'Crime Subcategory', 'Primary Offense Description', 'Precinct', 'Sector', 'Beat', 'Neighborhood')

11. View the results.
View(dfCrime)

12. You may also want to rename columns to make them more reader friendly or perhaps simplify the names. The select() function can be used to do this as well. Add the code you see below to see how this works. You simply pass in the new name of the column followed by an equal sign and then the old column name.

dfCrime = select(dfCrime, ‘CrimeDate’ = ‘Reported Date’, ‘Category’ = ‘Crime Subcategory’, ‘Description’ = ‘Primary Offense Description’, ‘Precinct’, ‘Sector’, ‘Beat’, ‘Neighborhood’)

Exercise 3: Filtering a dataset

In addition to limiting the columns that are part of a data frame, it’s also common to subset or filter the rows using a where clause. Filtering the dataset enables you to focus on a subset of the rows instead of the entire dataset. The dplyr package includes a filter() function that supports this capability. In this exercise you’ll filter the dataset so that only rows from a specific neighborhood are included.

1. In the RStudio **Console** pane add the following code. This will ensure that only crimes from the QUEEN ANNE neighborhood are included.
dfCrime2 = filter(dfCrime, Neighborhood == ‘QUEEN ANNE’)
2. Get the number of rows and view the data if you’d like with the View() function.
nrow(dfCrime2) [1] 25172

3. You can also include multiple expressions in a filter() function. For example, the line of code below would filter the data frame to include only residential burglaries that occurred in the Queen Anne neighborhood. There is no need to add the line of code below. It’s just meant as an example. We’ll examine more complex filter expressions in a later chapter.

dfCrime3 = filter(dfCrime, Neighborhood == ‘QUEEN ANNE’, Category == ‘BURGLARY-RESIDENTIAL’)

Exercise 4: Grouping and summarizing a dataset

The group_by() function, found in the dplyr package, is commonly used to group data by one or more variables. Once grouped, summary statistics can then be generated for the group or you can visualize the data in various ways. For example, the crime dataset we’re using in this chapter could be grouped by offense, neighborhood and year and then summary statistics including the count, mean, and median number of burglaries by year generated.

It’s also very common to visualize these grouped datasets in different ways. Bar

charts, scatterplots, or other graphs could be produced for the grouped dataset. In this exercise you'll learn how to group data and produce summary statistics.

1. In the RStudio console window add the code you see below to group the crimes by police beat.
dfCrime2 = group_by(dfCrime2, Beat)
2. The
n() function is used to get a count of the number of records for each group. Add the code you see below.
dfCrime2 = summarise(dfCrime2, n = n())
3. Use the head()
function to examine the results.
head(dfCrime2)

A tibble: 4 x 2 Beat n
<chr> <int>

1 D2 4373
2 Q1 88
3 Q2 10851
4 Q3 9860

Exercise 5: Plotting a dataset

The ggplot2 package can be used to create various types of charts and graphs from a data frame. The ggplot()function is used to define plots, and can be passed a number of parameters and joined with other functions to ultimately produce an output chart.

The first parameter passed to ggplot() will be the data frame you want to plot. Typically this will be a data frame object, but it can also be a subset of a data frame defined with the subset()function. The first code example on this slide passes a variable called housing, which contains a data frame. In the second code example, the subset() function is passed as the parameter. This subset function defines a filter that will include only rows where the State variable is equal to MA or TX.

In this exercise you will create a simple bar chart from the data frame created in the previous exercises in this chapter.

1. In the RStudio console add the code you see below. The ggplot() function in this case is passed the dfCrime data frame created in a previous exercises. The geom_col() function is used to define the geometry of the graph (bar chart) and is passed a mapping parameter which is defined by calling the aes() function and passing in the columns for the x axis (Beat), and the y axis (n = count).

ggplot(data=dfCrime2) + geom_col(mapping = aes(x=Beat, y=n), fill="red") 2.

This will produce the chart you see below in the Plots pane.

Exercise 6: Graphing burglaries by month and year

In this exercise we'll create something a little more complex. We'll create a couple bar charts that display the number of burglaries by year and by month for the Queen Anne neighborhood. In addition to the dplyr and ggplot2 packages we used previously in this chapter we'll also use the lubridate package to manipulate date information.

1. In the RStudio **Packages** pane, load the lubridate package. The lubridate package is part of tidyverse and is used to work with dates and times. Also, make sure the readr, dplyr and ggplot2 packages are loaded.

2. Load the crime data from the Crime_Data.csv
file.
dfCrime = read_csv("Crime_Data.csv", col_names = TRUE)
3. Specify the columns and column names.

dfCrime = select(dfCrime, 'CrimeDate' = 'Reported Date', 'Category' = 'Crime Subcategory', 'Description' = 'Primary Offense Description', 'Precinct', 'Sector', 'Beat', 'Neighborhood')

4. Filter the records so that only residential burglaries in the Queen Anne neighborhood are retained.
dfCrime2 = filter(dfCrime, Neighborhood == 'QUEEN ANNE', Category == 'BURGLARY-RESIDENTIAL')

5. The dplyr package includes the ability to dynamically create new columns in a data frame through the manipulation of data from existing columns in the data frame. The mutate() function is used to create the new columns. Here the mutate() function will be used to extract the year from the CrimeDate column.

Add the following code to see this in action. The second parameter creates a new column called YEAR and populates it by using the year() function from the lubridate package. Inside the year() function the CrimeDate column, which is a character column, is converted to a date and the format of the date

```
dfCrime3 = mutate(dfCrime2, YEAR = year(as. Date(dfCrime2$CrimeDate,
format='%m/%d/%Y')))
```

6. View the result. Notice the YEAR column at the end of the data frame. The mutate() function always adds new columns to the end of the data frame.

```
View(dfCrime3)
```

meDate	Category	Description	Precinct	Sector	Beat	Neighborhood	YEAR
22/2008	BURGLARY-RESIDENTIAL	BURGLARY-NOFORCE-RES	WEST	Q	Q3	QUEEN ANNE	2008
02/2008	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-RES	WEST	Q	Q3	QUEEN ANNE	2008
02/2008	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-RES	WEST	D	D2	QUEEN ANNE	2008
07/2008	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-RES	WEST	Q	Q3	QUEEN ANNE	2008
09/2008	BURGLARY-RESIDENTIAL	BURGLARY-NOFORCE-RES	WEST	Q	Q2	QUEEN ANNE	2008
10/2008	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-RES	WEST	Q	Q3	QUEEN ANNE	2008
19/2008	BURGLARY-RESIDENTIAL	BURGLARY-NOFORCE-RES	WEST	Q	Q2	QUEEN ANNE	2008
14/2008	BURGLARY-RESIDENTIAL	BURGLARY-NOFORCE-RES	WEST	Q	Q3	QUEEN ANNE	2008
17/2008	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-RES	WEST	Q	Q3	QUEEN ANNE	2008
17/2008	BURGLARY-RESIDENTIAL	BURGLARY-NOFORCE-RES	WEST	Q	Q2	QUEEN ANNE	2008
18/2008	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-RES	WEST	Q	Q2	QUEEN ANNE	2008
31/2008	BURGLARY-RESIDENTIAL	BURGLARY-NOFORCE-RES	WEST	D	D2	QUEEN ANNE	2008
30/2008	BURGLARY-RESIDENTIAL	BURGLARY-FORCE-RES	WEST	Q	Q3	QUEEN ANNE	2008

7. Now we'll group the data by year and summarize by getting a count of the number of crimes per year. Add the following lines of code.

```
dfCrime4 = group_by(dfCrime3, YEAR)
dfCrime4 = summarise(dfCrime4, n = n())
```

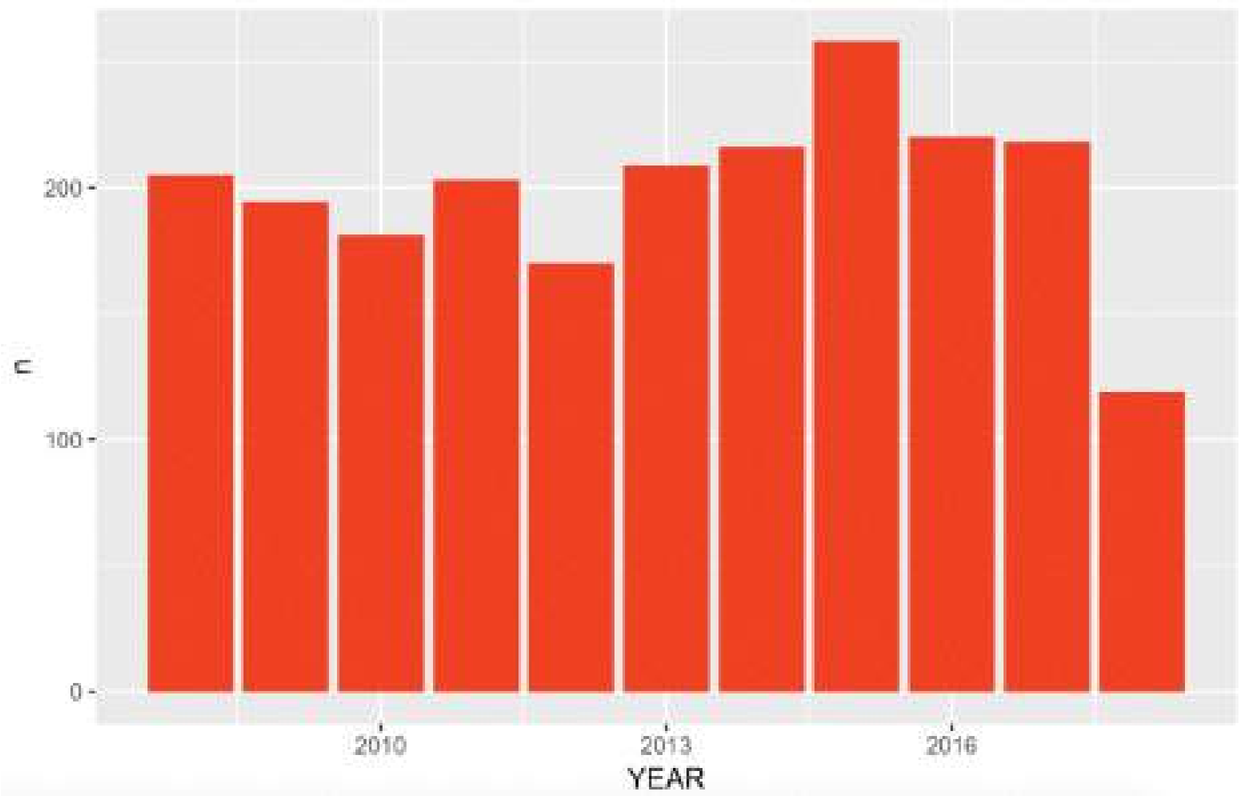
8. View the result.

```
View(dfCrime4)
```

	YEAR	n
1	2008	205
2	2009	194
3	2010	181
4	2011	203
5	2012	170
6	2013	209
7	2014	216
8	2015	258
9	2016	220
10	2017	218
11	2018	119

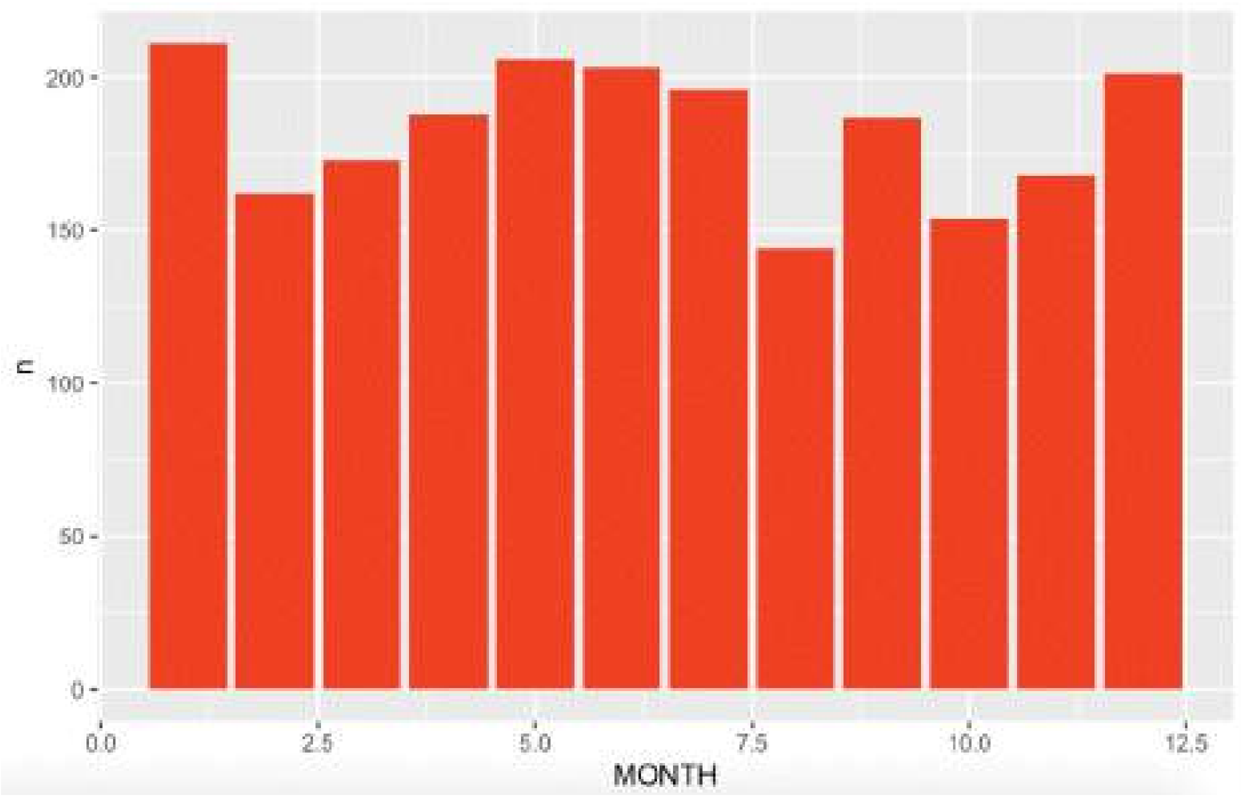
9. Create a bar chart by calling the ggplot() and geom_col() functions as seen below. Define YEAR as the column for the x axis and the number of crimes for the y axis. This should produce the chart you see below in the **Plots** pane.

```
ggplot(data=dfCrime4) + geom_col(mapping = aes(x=YEAR, y=n), fill=”red”)
```



10. Now we’ll create another bar chart that displays the number of crimes by month instead of year. First, create a MONTH column using the mutate() function.

dfCrime3 = mutate(dfCrime2, MONTH = month(as. Date(dfCrime2$CrimeDate, format=’%m/%d/%Y’)))
11. Group and summarize the data by month.
dfCrime4 = group_by(dfCrime3, MONTH) dfCrime4 = summarise(dfCrime4, n = n())
12. View the result.
View(dfCrime4)
13. Create the bar chart.
ggplot(data=dfCrime4) + geom_col(mapping = aes(x=MONTH, y=n), fill=”red”)



14. You can check your work against the solution file Chapter2_6.R.

Conclusion

In this chapter you learned some basic techniques for data exploration and visualization using the tidyversepackage and its ecosystem of sub-packages. After installing and loading the package using RStudio you performed a number of tasks using the R programming language with a number of tidyverse sub-packages. You loaded a dataset from a CSV file using readr. After, you manipulated the data in various ways using the dplyr package. The select() function was used to include and rename columns, and the contents of the data frame were filtered using the filter() function. The data was then grouped and summarized, and finally several graphs were produced using ggplot2.

In the next chapter you will learn how more about how to use the readr package to load data from external data sources.

Chapter 3

Loading Data into R

Large data objects, typically stored as data frames in R, are most often read from external files. R, along with tidyverse, include a number of functions that can read external data files from a wide variety of sources including text files of many varieties, relational databases, and web services. External text files need to have a specific format with the first line, called the header, containing the column names. Each additional line in the file will have values for each variable. In this chapter, we'll examine a number of functions that can be used to read data.

There are a number of common data formats that can be read into and out of R. This includes text files in formats such as csv, txt, html, and json. It also includes files output from statistical applications including SAS and SPSS. Online resources including web services and HTML pages can also be read into R. Finally, relational and non-relational database tables can be read as well. There are a number of functions provided by R and Tidyverse which will enable you to read these various sources.

In this chapter we'll cover the following topics:

- Loading a csv file with read.table()
- Loading a csv file with read.csv()
- Loading a tab delimited file with read.table()
- Using readr to load data

Exercise 1: Loading a csv file with read.table()

The first function we'll examine is read.table(). The read.table() function is a built in R function that can be used to read various file formats into a data frame. This is probably the most common internal function used for reading simple files into R. However, as we'll see later in the module, tidyverse includes similar functions which are actually more efficient at reading external data into R.

The syntax for read.table()is to accept a filename, which will be the path and file name, along with a TRUE|FALSE indicator for the header. If set to TRUE the assumption is that column names are in the header line of the file. The path is not necessary if you have already set the working directory. The output of the

read.table() function is a data frame object.

The header line, if included in the text file, will load a dataset into a data frame object. Default values will be used for the column headers if these are not provided. The file. choose()function is a handy function that you can use to interactively select the file you want imported rather than having to hard code the path to the dataset.

In this exercise you'll learn how to use the read.table() function to load a csv format file.
1. Open RStudio and find the **Console** pane.

2. If necessary, set the working directory by typing the code you see below into the **Console** pane or by going to **Session | Set Working Directory | Choose Directory** from the RStudio menu.

setwd(<installation directory for exercise data>)

3. The Data folder contains a file called StudyArea.csv, which is a comma separated file containing wildfire data from the years 1980-2016 for the states of California, Oregon, Washington, Idaho, Montana, Wyoming, Colorado, Utah, Nevada, Arizona, and New Mexico. There are a little over 439,000 records in this file and there are 37 columns of information that describe each fire during this period.

Use the read.table() function to load this data into a new data frame object. What happens when you run this line of code?
df = read.table("StudyArea.csv", header = TRUE)
You will get an error message when you attempt to run this line of code. The error message should appear as seen below.
Error in read.table("StudyArea.csv", header = TRUE) : more columns than column names

The reason an error message was generated in this case is that the read_ table() function uses spaces as the delimiter between records and our file uses commas as the delimiter.

4. Update your call to read.table() as seen below to include the sep argument, which should be a comma.
df = read.table("StudyArea.csv", sep=",", header = TRUE)

When you run this line of code you ‘ll see a new error.
Error in scan(file = file, what = what, sep = sep, quote = quote, dec = dec, :
line 12 did not have 14 elements

The read.table() function will NOT automatically fill in any missing values with a default value such as NA so because some of the columns are empty in our rows we get an error message that indicates a particular line didn’t have all 14 columns of information. We can fix this by adding the fill parameter and setting it equal to TRUE.

5. Update your code as seen below to add the fill parameter.
df = read.table(“StudyArea.csv”, header=TRUE, fill=TRUE, sep=”,”)

When you run this line of code it will import the contents of the file into a dataframe object. However, if you look at the **Environment** tab in R Studio you will see that it only loaded 153,095 records and yet we know there are over 400,000 records in the file. Quotes (single or double) in a csv file can cause records not to be loaded.

6. Let’s add one more parameter to handle records that were thrown out due to quotes.
df = read.table(“StudyArea.csv”, header=TRUE, fill=TRUE, quote=””, sep=”,”)

When you execute this line of code, 440,476 records should be imported. The data is loaded into an R dataframe object which is a structure that resembles a table. Detailed information about dataframe objects will be covered in a later section of the course. For now, you can think of them as tables containing columns and rows.

My point in showing you this is to show how difficult it can be to use the read.table() function to load the contents of a csv file. The read.table() function is typically used to load tab delimited text files, but many people will attempt to use the read.table() function with csv format files without understanding all the parameters that may need to be included. Instead, you should use read.csv() as we’ll do in the next step.

7. You can check your work against the solution file Chapter3_1.R.

Exercise 2: Loading a csv file with read.csv()

The read.csv()function is also a built in R function that is almost identical to read. table(), with the exception that the header and fill arguments are set to TRUE by default. In this step you'll see how much easier it is to load a csv file using read.csv().

1. The read.csv() function automatically handles most of the situations you are required to identify when using read.table() to load a csv file. Enter and run the code you see below to see how much easier this is with read.csv().

df = read.csv("StudyArea.csv")

2. This will correctly load all 400,000+ records from the csv file! See how much easier that is? There will be a few records missing, but overall this function is much easier to use than read.table().

3. You can check your work against the solution file Chapter3_2.R.

Exercise 3: Loading a tab delimited file with read.table()

The read.table() function is most often used to read the contents of a tab delimited file. In this step you'll learn how to do that.

1. Your Data folder includes a file called all_genes_pombase.txt, which is text delimited. Open this file with Excel or some other application to see the field structure and delimiters.

2. In the R Console window enter and run the code you see below to import the file.
df2 = read.table("all_genes_pombase.txt", header=TRUE, sep="\t", quote="")

3. This should load 7019 records into the dataframe. You'll notice that many of the parameters still need to be used when loading the dataset so it's not as easy to use as you might hope even in this case.

4. You can check your work against the solution file Chapter3_3.R.

Exercise 4: Using readr to load data

So far in this chapter we've been looking at various built in R functions for

reading external files into R as data frames. The tidyverse package includes a sub-package called readr that can also be used to load external data. The readr package includes a read_csv() function that loads data much faster than the internal read.csv() function.

In addition to loading the data faster it also includes a progress dialog and the output includes the data frame column structure along with any parsing errors. Overall, the read_csv() function in the readr package is preferred over the functions found in the basic installation of R. The readr package also includes some other functions for loading various file formats including read_delim(), read_csv2(), and read_tsv(). Each of the functions accept the same parameters, so once you've learned to use any of the R functions for loading data you can easily use any of the others.

In this step you're going to use the read_csv() function found in the readr package to load data into a data frame.

1. Load the readr library.

```
library(readr)
```

2. The read_csv() function in the readr package can be used to load csv files. Compared to the base loading functions we looked at previously in this exercise, readr functions are significantly faster (10x), include a helpful progress bar to provide feedback on the progress of the load for large files, and all the functions work exactly the same way.

Add and run the code you see below. Notice how much more quickly the data loads into the dataframe object. The col_types argument was used in this case to load all the columns as a character data type for simplification purposes. Otherwise we'd have to do some additional preprocessing of the data to account for various column data types.

```
dfReadr = read_csv("StudyArea.csv", col_types = cols(.default = "c"),
col_names = TRUE)
```

Other loading functions found in the readr package include read_ delim(), read_csv2(), read_tsv()

3. Now let's run this function again, but this time take off the col_types argument so you can see an example of some of the potential loading errors that can occur. Update and run your code as follows:

dfReadr = read_csv(“StudyArea.csv”, col_names = TRUE)
4. The first thing you’ll see is a list of the columns that will be imported along with the column data type. Your output should appear as follows:
Parsed with column specification:

cols(
.default = col_character(), FID = col_integer(),
UNIT = col_integer(),
FIRENUMBER = col_integer(), SPECCAUSE = col_integer(), STATCAUSE = col_integer(), SIZECLASSN = col_integer(), FIRETYPE = col_integer(), PROTECTION = col_integer(), FIREPROTTY = col_integer(), YEAR_ = col_integer(), FiscalYear = col_integer(), STATE_FIPS = col_integer(), FIPS = col_integer(),
DLATITUDE = col_double(), DLONGITUDE = col_double(), TOTALACRES = col_double(), TRPGENCAUS = col_integer(), TRPSPECCAU = col_integer(), Duplicate_ = col_integer()

)
5. A warning message will be displayed below that indicating that there were parsing errors on the load.
Warning: 196742 parsing failures.
row # A tibble: 5 x 5 col
row col expected actual file expected

<int> <chr> <chr> <chr> <chr> actual 1 242621 UNIT an integer EOR ‘StudyArea.csv’ file 2 242622 UNIT an integer EOR ‘StudyArea.csv’ row 3 242623 UNIT an integer EOR ‘StudyArea.csv’ col 4 242624 UNIT an integer EOR ‘StudyArea.csv’ expected

5 242625 UNIT an integer EOR ‘StudyArea.csv’
6. You can use the
problems()
function to get a list of the parsing errors. Add and run the code you see below.
problems(dfReadr)
# A tibble: 196,742 x 5
row col expected actual file

<int> <chr> <chr> <chr> <chr>
1 242621 UNIT an integer EOR ‘StudyArea.csv’

2 242622 UNIT an integer EOR ‘StudyArea.csv’
3 242623 UNIT an integer EOR ‘StudyArea.csv’
4 242624 UNIT an integer EOR ‘StudyArea.csv’
5 242625 UNIT an integer EOR ‘StudyArea.csv’
6 242626 UNIT an integer EOR ‘StudyArea.csv’
7 242627 UNIT an integer EOR ‘StudyArea.csv’
8 242628 UNIT an integer EOR ‘StudyArea.csv’
9 242629 UNIT an integer EOR ‘StudyArea.csv’ 10 242630 UNIT an integer EOR ‘StudyArea.csv’ # ... with 196,732 more rows

7. From the looks of the error messages it appears there is an issue with the UNIT column. If you look back up to the list of columns and data types, you’ll notice that the UNIT column was created as an integer data type. However, if you open the StudyArea.csv file in Excel or another application you’ll quickly see that not all the values are numeric. Some include letters. This accounts for the parsing errors in the dataset.

Update your code as seen below and run it again. This sets the UNIT column to a character (text) data type.
dfReadr = read_csv(“StudyArea.csv”, col_types = list(UNIT = col_ character()), col_names = TRUE)

This time you should get a clean load of the dataset. That doesn’t mean the data won’t need some additional preparation and cleanup. For example, there are some date fields including STARTDATED that were loaded as character but might be better off as date fields. We can save this additional preparation work for a later exercise though.

8. You can examine the first few lines of the dataframe by entering the head() function as seen below.
head(dfReadr)
# A tibble: 6 x 14
FID ORGANIZATI UNIT SUBUNIT SUBUNIT2 FIRENAME CAUSE YEAR_

STARTDATED CONTRDATED OUTDATED STATE STATE_FIPS
<int> <chr> <chr> <chr> <chr> <chr> <chr> <int>
<chr> <chr> <chr> <chr> <int>
1 0 FWS 81682 USCADBR San Diego Bay… PUMP HOU… Human 2001

1/1/01 0:00 1/1/01 0:… NA Cali… 6
2 1 FWS 81682 USCADBR San Diego Bay… I5 Human 2002
5/3/02 0:00 5/3/02 0:… NA Cali… 6
3 2 FWS 81682 USCADBR San Diego Bay… SOUTHBAY Human 2002
6/1/02 0:00 6/1/02 0:… NA Cali… 6
4 3 FWS 81682 USCADBR San Diego Bay… MARINA Human 2001
7/12/01 0:… 7/12/01 0… NA Cali… 6
5 4 FWS 81682 USCADBR San Diego Bay… HILL Human 1994
9/13/94 0:… 9/13/94 0… NA Cali… 6
6 5 FWS 81682 USCADBR San Diego Bay… IRRIGATI… Human 1994
4/22/94 0:… 4/22/94 0… NA Cali… 6
... with 1 more variable: TOTALACRES <dbl>

9. You can check your work against the solution file
Chapter3_4.R.

Conclusion

In this chapter you learned various functions for loading an external data file including the built in R functions read.table() and read.csv(). While these functions can certainly get the job done, the read_csv() function found in the readr package is a much more efficient function for loading external data.
In the next chapter you will learn how to transform your datasets using the dplyr package. You'll learn techniques for filtering the contents of a data frame, selecting specific columns to be used, arranging rows in ascending or descending order, and summarize and group a dataset.

Chapter 4

Transforming Data

Before a dataset can be analyzed in R it often needs to be manipulated or transformed in various ways. The dplyr package, part of the larger tidyverse package, provides a set of functions that allow you to transform a dataset in various ways. The dplyr package is a very important part of tidyverse since the functions provided through this package are used so frequently to transform data in different ways prior to doing more advanced data exploration, visualization, and modeling.

There are five key functions that are part of dplyr: filter(), arrange(), select(), mutate(), and summarize().All five functions work in a similar manner where the first argument is the data frame to manipulate, the next N number of parameters defined the columns to include, and all return a data frame as a result.

The dplyr functions are often used in conjunction with the group_by() dplyr function to manipulate a dataset that has been grouped in some way. The group_by() function creates a new data frame object that has been grouped by one or more variables.

In this chapter we'll cover the following topics:

- Filtering records to create a subset
- Narrowing the list of columns
- Arranging rows in ascending or descending order
- Adding rows
- Summarizing and grouping
- Piping for code efficiency

Exercise 1: Filtering records to create a subset

The first dplyrfunction that we'll examine is filter(). The filter() function is used to create a subset of records based on some value. For example, you might want to create a data frame of wildfires containing incidents that have burned more than 25,000 acres. As long as you have an existing data frame that includes a column that measures the number of acres burned, you can accomplish the creation of this subset using the filter() function.

As will be the case with all the dplyrfunctions we examine, the first argument passed to the filter() function is a data frame object. Each additional parameter passed to the function is a conditional expression used to filter the data frame. For example, take a look at the line of code below. This statement calls the filter() function to create a new variable called df25k, which will contain only rows where the ACRES column contains a value greater than 25000.

```
df25k = filter(df, ACRES >= 25000)
```

This is an example of calling the filter() function and passing a single conditional expression. In the next code example, two conditional expressions are passed. The first is used to filter records so that the number of acres is greater than or equal to 25000, and the second filter records so that only records where the Year column contains a value of 2016 will be retained.

```
df25k = filter(df, ACRES >= 25000, YEAR == 2016)
```

In this case, the df25k variable will include records where both conditions are matched: acreage burned is greater than 25000 and the fire year was 2016. This can also be rewritten as a single parameter that uses the & operator to combine expressions as seen below.

```
df25k = filter(df, ACRES >= 25000 & YEAR == 2016)
```

In this exercise you'll learn how to use the filter() function to create a subset of records based on some value.

1. The exercises in this chapter require the following packages: readr, dplyr, ggplot2. They can be loaded from the **Packages** pane, the **Console** pane, or a script.

2. Open RStudio and find the **Console** pane.

3. If necessary, set the working directory by typing the code you see below into the **Console** pane or by going to **Session | Set Working Directory | Choose Directory** from the RStudio menu.

```
setwd(<installation directory for exercise data>)
```

4. The Data folder contains a file called StudyArea.csv, which is a comma separated file containing wildfire data from the years 1980-2016 for the states of

California, Oregon, Washington, Idaho, Montana, Wyoming, Colorado, Utah, Nevada, Arizona, and New Mexico. There are a little over 439,000 records in this file and there are 37 columns of information that describe each fire during this period.

Use the read_csv() function to load the dataset into a data frame.

```
dfFires = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
```

5. Use the nrow() function to make sure that the approximately 439,000 records were loaded.

```
nrow(dfFires)
[1] 439362
```

6. Initially we'll use a single conditional expression with the filter() function to create a subset of records that contains only wildfires that are greater than 25,000 acres. Add the code you see below to run the filter() function. All dplyr functions, including filter(), return a new data frame object so you need to specify a new variable that will contain the output data frame. The df25k variable will hold the output data frame in this case.

```
df25k = filter(dfFires, TOTALACRES >= 25000)
```

Get a count of the number of records that match the filter. There should be 655 rows. You may also want to use the View(df25k) function to see the data in a tabular format.

```
nrow(df25k)
[1] 655
```

7. You can also include multiple conditional expressions as part of the filter. Each expression (argument) is combined with an "and" clause by default. This means that all expressions must be matched for a recorded to be returned. Add and run the code you see below to see an example.

```
df1k = filter(dfFires, TOTALACRES >= 1000, YEAR_ == 2016)
nrow(df1k)
[1] 152
```

8. You can also combine the expressions into a single expression with multiple conditions as seen below. This will accomplish the same thing as the previous

line of code. Which of the two you use is a matter of personal preference in this case since we're using an "and" clause. The & character is the "and" operator. You would need to use the | character to include an "or" operator.

```
df1k = filter(dfFires, TOTALACRES >= 1000 & YEAR_ == 2016)
```

9. Finally, when you have a list of potential values that you want to be included by the filter the %in% statement can be used. Add the line of code below to see how this works. This particular line of code would create a data frame containing fires that occurred in the years 2010, 2011, or 2012.

```
dfYear = filter(dfFires,YEAR_ %in% c(2010, 2011, 2012))
```

10. You can view any of these data frames in a tabular view using the View(<data frame>) syntax. For example,
View(dfYear) 11. You can check your work against the solution file Chapter4_1.R.

Exercise 2: Narrowing the list of columns with select()

Many datasets that you load from external data sources include dozens of columns. The StudyArea.csvfile that you've been working with in the exercises includes 37 columns of information. In most cases you won't need all the columns.

The select() function can be used to narrow down the list of columns to include only those needed for a task. To use the select() function, simply pass in the name of the data frame along with the columns to include.

1. Use the read_csv() function to load the dataset into a data frame.

Note: For the sake of completeness you will be loading the external data from the StudyArea.csv file to the dfFires data frame, but this step isn't absolutely necessary if you're doing the exercises in sequence in the same R Studio session.

```
dfFires = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()),
col_names = TRUE)
```

2. On a new line, add a call to the select()
function as seen below to limit the columns that are returned.

```
dfFires2 = select(dfFires, FIRENAME, TOTALACRES, YEAR_)
```

3. Display the first few rows and notice that we now have only three columns.
head(dfFires2)

FIRENAME TOTALACRES YEAR_ <chr> <dbl> <int> 1 PUMP HOUSE 0.100 2001 2 I5 3.00 2002 3 SOUTHBAY 0.500 2002 4 MARINA 0.100 2001 5 HILL 1.00 1994 6 IRRIGATION 0.100 1994

4. Many of the column names that you import will not be very reader friendly so it's not uncommon to want to rename the columns as well. This can be accomplished using the select() function as well. Rename your columns by adding and running the code you see below.

dfFires2 = select(dfFires, “FIRE” = “FIRENAME”, “ACRES” = “TOTALACRES”, “YR” = “YEAR_”)
5. Display the first few lines.
head(dfFires2)

FIRE ACRES YR <chr> <dbl> <int> 1 PUMP HOUSE 0.100 2001 2 I5 3.00 2002 3 SOUTHBAY 0.500 2002 4 MARINA 0.100 2001 5 HILL 1.00 1994 6 IRRIGATION 0.100 1994

6. There are also a number of handy helper functions that you can use with the select() function to filter the returned columns. These include starts_with(), ends_with(), contains(), matches(), and num_range(). To see how this works, add and run the code you see below. This will return any columns that contain the word DATE.

dfFires3 = select(dfFires, contains(“DATE”))
head(dfFires3)

STARTDATED CONTRDATED OUTDATED <chr> <chr> <chr> 1 1/1/01 0:00 1/1/01 0:00 NA 2 5/3/02 0:00 5/3/02 0:00 NA 3 6/1/02 0:00 6/1/02 0:00 NA 4 7/12/01 0:00 7/12/01 0:00 NA 5 9/13/94 0:00 9/13/94 0:00 NA 6 4/22/94 0:00 4/22/94 0:00 NA

7. You can also make multiple calls to these helper functions.
dfFires3 = select(dfFires, contains(“DATE”), starts_with(“TOTAL”))
head(dfFires3)

D STARTDATED CONTRDATED OUTDATED TOTALACRES <chr> <chr>

<chr> <dbl>
1 1/1/01 0:00 1/1/01 0:00 NA 0.100
2 5/3/02 0:00 5/3/02 0:00 NA 3.00
3 6/1/02 0:00 6/1/02 0:00 NA 0.500
4 7/12/01 0:00 7/12/01 0:00 NA 0.100
5 9/13/94 0:00 9/13/94 0:00 NA 1.00 6 4/22/94 0:00 4/22/94 0:00 NA 0.100

8. You can check your work against the solution file
Chapter4_2.R.

Exercise 3: Arranging Rows

The arrange() function in the dplyr package can be used to order the rows in a data frame. This function accepts a set of columns to order by with the default row ordering being in ascending order. However, you can pass the desc() helper function to order the rows in descending order. Missing values will be placed at the end of the data frame.

1. Use the read_csv()
function to load the dataset into a data frame.
dfFires = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
2. Filter the dataset so that it contains only fires greater than 1,000 acres burned from the year 2016.
df1k = filter(dfFires, TOTALACRES >= 1000, YEAR_ == 2016)
3. Add and run the code you see below to create a subset of columns and rename them.
df1k = select(df1k, "NAME" = "FIRENAME", "ACRES" = "TOTALACRES", "YR" = "YEAR_")
4. Sort the rows so that they are in ascending order.
arrange(df1k, ACRES)

NAME ACRES YR <chr> <dbl> <int> 1 Crackerbox 1000. 2016 2 Lakes 1000. 2016 3 Choulic 2 1008. 2016 4 Amigo Wash 1020. 2016 5 Granite 1030. 2016 6 Tie 1031. 2016 7 Black 1040. 2016 8 Bybee Creek 1072. 2016 9 MARSHES 1080. 2016 10 Bug Creek 1089. 2016

5. Use the
desc()

helper function to order the rows in descending order.
arrange(df1k, desc(ACRES))

NAME ACRES YR <chr> <dbl> <int> 1 PIONEER 188404. 2016 2 Junkins 181320. 2016 3 Range 12 171915. 2016 4 Erskine 48007. 2016 5 Cedar 45977. 2016 6 Maple 45425. 2016 7 Rail 43799. 2016 8 North Fire 42102. 2016 9 Laidlaw 39813. 2016 10 BLUE CUT 36274. 2016

6. You can use the
View() function as a wrapper around these calls to view the data in a tabular grid view by adding the code you see below.
View(arrange(df1k, desc(ACRES))) 7. You can check your work against the solution file Chapter4_3.R.

Exercise 4: Adding Rows with mutate()

The mutate() function is used to add new columns to a data frame that are the result of a function you run on other columns in the data frame. Any new columns created with the mutate() function will be added to the end of the data frame. This function can be incredibly useful for dynamically creating new columns that are the result of operations performed on other columns from the data frame. In this exercise you'll learn how the mutate() function can be used to create new columns in a data frame.

1. You're going to need the lubridate package for this exercise. The lubridate package is part of tidyverse and is used to work with dates and times. In R Studio, check the **Packages** tab to make sure that lubridate has been installed and loaded as seen in the screenshot below. If not, you'll need to do so now using the instructions for installing and loading a package covered in *Chapter 1: Introduction to R*.

2. Recall from *Chapter 1: Introduction to R* that you can also load an installed library using the syntax seen below.
library(lubridate)
3. Use the read_csv() function to load the dataset into a data frame.
dfFires = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
4. Use the select()
function to define a set of columns for the data frame.

df = select(dfFires, ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE, STARTDATED)

5. Do some basic filtering of the data so that only fires greater than 1,000 acres burned and have a cause of Human or Natural are included. There are some records marked as Unknown in the dataset, so we'll remove those for this exercise.

df = filter(df, TOTALACRES >= 1000 & CAUSE %in% c('Human', 'Natural'))

6. Use the mutate() function to create a new DOY column that contains the day of the year that the fire started. The yday() function from the lubridate package is used to return the day of the year using a formatted input date from the STARTDATED column.

df = mutate(df, DOY = yday(as.Date(df$STARTDATED, format='%m/%d/%y %H:%M')))

7. View the resulting DOY column.

View(df)

	ORGANIZATI	STATE	YEAR_	TOTALACRES	CAUSE	STARTDATED	DOY
1	FWS	Arizona	1988	1500.0	Human	3/26/88 0:00	86
2	FWS	Arizona	1986	10390.0	Human	5/15/86 0:00	135
3	FWS	Montana	1986	1400.0	Human	6/27/86 0:00	178
4	FWS	Arizona	2002	1035.0	Human	2/28/02 0:00	59
5	FWS	Arizona	2000	5700.0	Human	4/9/00 0:00	100
6	FWS	Arizona	2000	2750.0	Human	5/14/00 0:00	135
7	FWS	Arizona	2002	5312.0	Human	5/14/02 0:00	134
8	FWS	Arizona	2000	5200.0	Human	6/2/00 0:00	154
9	FWS	Arizona	1991	6530.0	Human	5/16/91 0:00	136
10	FWS	Arizona	1991	2500.0	Natural	7/26/91 0:00	207
11	FWS	Arizona	1994	2500.0	Human	7/6/94 0:00	187
12	FWS	Arizona	1994	1200.0	Human	7/4/94 0:00	185
13	FWS	California	1992	1800.0	Human	3/18/92 0:00	78

8. You can check your work against the solution file Chapter4_4.R.

9. In the next exercise the mutate() function will be used again when we create a column that holds the decade of the fire and then calculates the total acreage burned by acreage.

Exercise 5: Summarizing and Grouping

Summary statistics for a data frame can be produced with the summarize() function. The summarize() function produces a single row of data containing summary statistics from a data frame. This function is normally paired with the group_by() function to produce group summary statistics.

The grouping of data in a data frame facilitates the split-apply-combine paradigm. This paradigm first splits the data into groups, using the group_by() function in dplyr, then applies analysis to the group, and finally, combines the results. The group_by() function handles the split portion of the paradigm by creating groups of data using one or more columns. For example, you might group all wildfires by state and cause.

In this step you'll use the mutate(), summarize(), and group_by() functions to group wildfires by decade and produce a summary of the mean wildfire size for each decade.
1. Use the read_csv() function to load the dataset into a data frame.

dfFires = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
2. Select the columns that will be used in the exercise.
df = select(dfFires, ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE)
3. Filter the records.
df = filter(df, TOTALACRES >= 1000)

4. Use the mutate() function to create a new column called DECADE that defines the decade in which each fire occurred. In this case an ifelse() function is called to produce the values for each decade.

function is called to produce the values for each decade.

1989", ifelse(YEAR_ %in% 1990:1999, "1990-1999", ifelse(YEAR_ %in% 2000:2009, "2000-2009", ifelse(YEAR_ %in% 2010:2016, "2010-2016", "-99"))))) 5. View the result.
View(df)

	ORGANIZATI	STATE	YEAR_	TOTALACRES	CAUSE	DECADE
1	FWS	Arizona	1988	1500.0	Human	1980-1989
2	FWS	Arizona	1986	10390.0	Human	1980-1989
3	FWS	Montana	1986	1400.0	Human	1980-1989
4	FWS	Arizona	2002	1035.0	Human	2000-2009
5	FWS	Arizona	2000	5700.0	Human	2000-2009
6	FWS	Arizona	2000	2750.0	Human	2000-2009
7	FWS	Arizona	2002	5312.0	Human	2000-2009

6. Use the group_by() function to group the data frame by decade.

```
grp = group_by(df, DECADE)
```

7. Summarize the mean size of wildfires by decade using the summarize() function.

```
sm = summarize(grp, mean(TOTALACRES))
```

8. View the result.

```
View(sm)
```

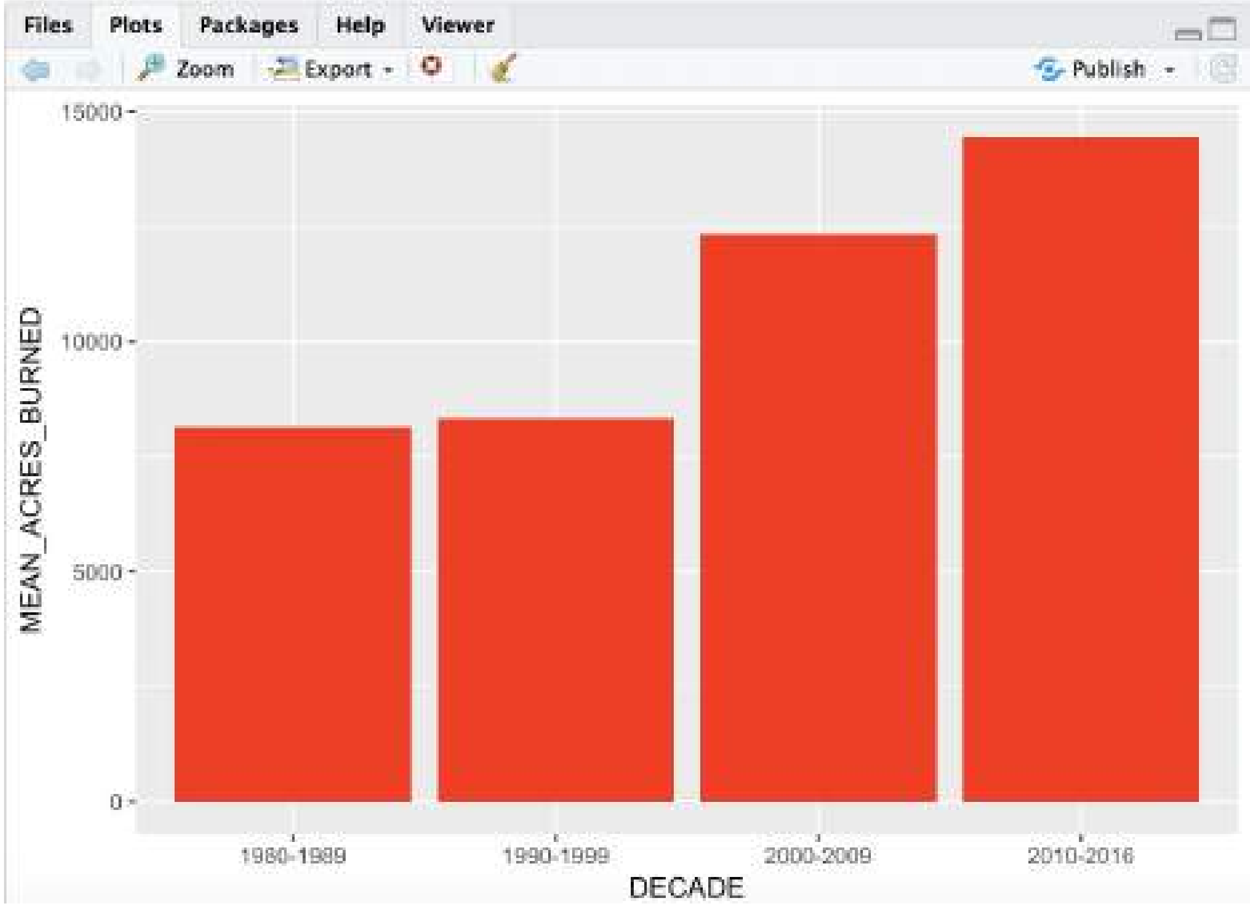
	DECADE	mean(TOTALACRES)
1	1980-1989	8128.645
2	1990-1999	8333.036
3	2000-2009	12329.181
4	2010-2016	14443.197

9. Let’s tidy things up by renaming the new column produced by the summarize() function.

```
names(sm) <- c(“DECADE”, “MEAN_ACRES_BURNED”)
```

10. Finally, let’s create a bar chart of the results. We’ll discuss the creation of many different types of charts and graphs as we move through later chapters of the book so detailed discussion of these topics will be saved for later.

```
ggplot(data=sm) + geom_col(mapping = aes(x=DECADE, y=MEAN_ACRES_BURNED), fill=”red”)
```



11. You can check your work against the solution file Chapter4_5.R.

Exercise 6: Piping

As you've probably noticed in some of these exercises, it is not unusual to run a series of dplyrfunctions as part of a larger processing routine. As you'll recall, each dplyr function returns a new data frame, and this data frame is typically used as the input to the next dplyr function in the series. These data frames are intermediate datasets not needed beyond the current step. However, you are still required to name and code each of these datasets.

Piping is a more efficient way of handling these temporary, intermediate datasets. In sum, piping is an efficient way of sending the output of one function to another function without creating an intermediate dataset and is most useful when you have a series of functions to run. The syntax for piping is to use the %>% characters at the end of each statement that you want to pipe. In this exercise you'll learn how to use piping to chain together input and output data frames.

1. In the last exercise the select(), filter(), mutate(), group_by(), and summarize() function were all used in a series that ultimately produced a bar chart showing the mean acreage burned by wildfires in the past few decades. Each of these functions return a data frame, which is then used as input to the next function in the series. Piping is a more efficient way of coding this chaining of function calls. Rewrite the code produced in *Exercise 4: Adding Rows with mutate()* as seen below and then we'll discuss how piping works.

```
library(lubridate)
df = read_csv("StudyArea.csv", col_types = list(UNIT = col_character()),
col_names = TRUE) %>%
select(ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE,
STARTDATED) %>%
filter(TOTALACRES >= 1000 & CAUSE %in% c('Human', 'Natural')) %>%
mutate(DOY = yday(as.Date(STARTDATED, format='%m/%d/%y %H:%M')))
View(df)
```

The first line of code reads the contents of the external StudyArea.csv file into a data frame variable (df) as we've done in all the other exercises in this chapter. However, you'll notice the inclusion of the piping statement (%>%>) at the end of the line. This ensures that the contents of the df variable will automatically be sent to the select() function.

Notice that the select() function does not create a variable like we have done in the past exercises, and that we have left off the first parameter, which would normally have been the data frame variable. It is implied that the df variable will be passed to the select() function.
This same process of including the piping statement at the end of each line and leaving off the first parameter is repeated for all the additional lines of code where we want to automatically pass the df variable to the next dplyr function. Finally, we view the contents of the df variable using the View() function on the last line.

Piping makes your code more streamlined and easier to read and also takes away the need to create and populate variables that are only used as intermediate datasets.

2. You can check your work against the solution file Chapter4_6.R.

Exercise 7: Challenge

The challenge step is optional, but it will give you a chance to reinforce what you've learned in this module. Create a new data frame that is a subset of the original dfFires data frame. The subset should contain all fires from the State of Idaho and the columns should be limited so that only the YEAR_, CAUSE, and TOTALACRES columns are present. Rename the columns if you wish. Group the data by CAUSE and YEARand then summarize by total acres burned. Plot the results.

Conclusion

In this chapter you learned how to use the dplyr package to perform various data transformation functions. You learned how to limit columns with the select() function, filter a data frame based on one or more expressions, add columns with mutate(), and summarize and group data. Finally, you learned how to use piping to make your code more efficient.

In the next chapter you'll how to create tidy datasets with the tidyr package.
Chapter 5

Creating Tidy Data

Let’s first describe what we mean by “tidy data”, because the term doesn’t necessarily fully describe the concept. Data tidying is a consistent way of organizing data in R and can be facilitated through the tidyr package found in the tidyverse ecosystem. There are three rules that we can follow to make a dataset tidy. First, each variable must have its own column. Second, each observation must have its own row, and finally, each value must have its own cell. This is illustrated by the diagram below.

There are two main advantages of having tidy data. One is more of a general advantage and the other is more specific. First, having a consistent, uniform data structure is very important. The other packages that are part of tidyverse, including dplyr and ggplot2 are designed to work with tidy data so ensuring that your data is uniform facilitates the efficient processing of your data. In addition, placing variables into columns allows for the easily facilitation of vectorization in R.

Many datasets that you encounter will not be tidy and will require some work on your end. There can be many reasons why a dataset isn’t tidy. Oftentimes the people who created the dataset aren’t familiar with the principles of tidy data. Unless you are trained in the practice of creating tidy datasets or spend a lot of time working with data structures these concepts aren’t readily apparent. Another common reason that datasets aren’t tidy is that data is often organized to facilitate something other than analysis. Data entry is perhaps the most common of the reasons that fall into this category. To make data entry as easy as possible, people will often arrange data in ways that aren’t tidy. So, many datasets require some sort of tidying before you can begin your analysis.

The first step is to figure out what the variables and observations are for the dataset. This will facilitate your understanding of what the columns and rows should be. In addition, you will also need to resolve one or two common problems. You will need to figure out if one variable is spread across multiple columns, and you will need to figure out if one observation is scattered across multiple rows. These concepts are known as gathering and spreading. We'll examine these concepts further in the exercises in this chapter.

In this chapter we'll cover the following topics:

- Gathering
- Spreading
- Separating
- Uniting

Exercise 1: Gathering

A common problem in many datasets is that the column names are not variables but rather values of a variable. In the figure below, the 1999 and 2000 columns are actually values of the variable YEAR. Each row in the existing table actually represents two observations. The tidyrpackage can be used to gather these existing columns into a new variable. In this case, we need to create a new column called YEARand then gather the existing values in the 1999 and 2000 columns into the new YEAR column.

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

The gather() function from the tidyr package can be used to accomplish the gathering of data. Take a look at the line of code below to see how this function works.

gather('1999', '2000', key = 'year', value = 'cases')

There are three parameters of the gather()function. The first is the set of columns that represent what should be values and not variables. These would be the 1999 and 2000 columns in the example we have been following. Next, you'll need to name the variable of the new column. This is also called the key, and in this case will the year variable. Finally, you'll need to provide the value, which is the name of the variable whose values are spread over the cells.

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

table4

In this exercise you'll learn how to use the gather() function to resolve the types of problems we discussed in the introduction to this topic.

1. In the Data folder where you installed the exercise data for this book is a file called CountryPopulation.csv. Open this file, preferably in Microsoft Excel, or some other type of spreadsheet software. The file should look similar to the screenshot below. This spreadsheet includes should look similar to the screenshot below. This spreadsheet includes 2017. The columns for each year represent values, not variables. These columns need to be gathered into a new pair of variables that represent the Year and Population. In this exercise you'll use the gather() function to accomplish this data tidying task.

1	Country Name	Country Code	2010	2011	2012	2013	2014	2015	2016	2017
2	Aruba	ABW	101669	102053	102577	103187	103795	104341	104822	105264
3	Afghanistan	AFG	28803167	29708599	30696958	31731688	32758020	33736494	34656032	35530081
4	Angola	AGO	23369131	24218565	25096150	25998340	26920466	27859305	28813463	29784193
5	Albania	ALB	2913021	2905195	2900401	2895092	2889104	2880703	2876101	2873457
6	Andorra	AND	84449	83751	82431	80788	79223	78014	77281	76965
7	Arab World	ARB	356508908	364895878	373306993	381702086	390043028	398304960	406452690	414491886
8	United Arab Emirates	ARE	8270684	8672475	8900453	9006263	9070867	9154302	9269612	9400145
9	Argentina	ARG	41223889	41656879	42096739	42539925	42981515	43417765	43847430	44271041
10	Armenia	ARM	2877311	2875581	2881922	2893509	2906220	2916950	2924816	2930450
11	American Samoa	ASM	55637	55320	55230	55307	55437	55537	55599	55641
12	Antigua and Barbuda	ATG	94661	95719	96777	97824	98875	99923	100963	102012
13	Australia	AUS	22031750	22340024	22742475	23145901	23504138	23850784	24210809	24598933
14	Austria	AUT	8363404	8391643	8429991	8479823	8546356	8642699	8736668	8809212
15	Azerbaijan	AZE	9054332	9173082	9295784	9416801	9535079	9649341	9757812	9862429
16	Burundi	BDI	8766930	9043508	9319710	9600186	9891790	10199270	10524117	10864245

2. Open RStudio and find the **Console** pane.

3. If necessary, set the working directory by typing the code you see below into the **Console** pane or by going to **Session | Set Working Directory | Choose Directory** from the RStudio menu.

```
setwd(<installation directory for exercise data>)
```

4. If necessary, load the readr and tidyr packages by clicking the check boxes in the **Packages** pane or by including the following line of code.

```
library(readr)
library(tidyr)
```

5. Load the CountryPopulation.csv file into RStudio by writing the code you see below in the **Console** pane.

```
dfPop = read_csv("CountryPopulation.csv", col_names = TRUE)
```

You should see the following output in the **Console** pane.

```
Parsed with column specification:
cols(

`Country Name` = col_character(),
`Country Code` = col_character(), `2010` = col_double(),
`2011` = col_double(),
`2012` = col_double(),
`2013` = col_double(),
`2014` = col_double(),
`2015` = col_double(),
`2016` = col_double(),
`2017` = col_double()
```

)
6. Use the
View() function to display the data in a tabular structure.
View(dfPop)

	Country Name	Country Code	2010	2011	2012	2013
1	Aruba	ABW	101669	102053	102577	103187
2	Afghanistan	AFG	28803167	29708599	30696958	31731688
3	Angola	AGO	23369131	24218565	25096150	25998340
4	Albania	ALB	2913021	2905195	2900401	2895092
5	Andorra	AND	84449	83751	82431	80788
6	Arab World	ARB	356508908	364895878	373306993	381702086
7	United Arab Emirates	ARE	8270684	8672475	8900453	9006263
8	Argentina	ARG	41223889	41656879	42096739	42539925
9	Armenia	ARM	2877311	2875581	2881922	2893509
10	American Samoa	ASM	55637	55320	55230	55307
11	Antigua and Barbuda	ATG	94661	95719	96777	97824
12	Australia	AUS	22031750	22340024	22742475	23145901
13	Austria	AUT	8363404	8391643	8429991	8479823
14	Azerbaijan	AZE	9054332	9173082	9295784	9416801

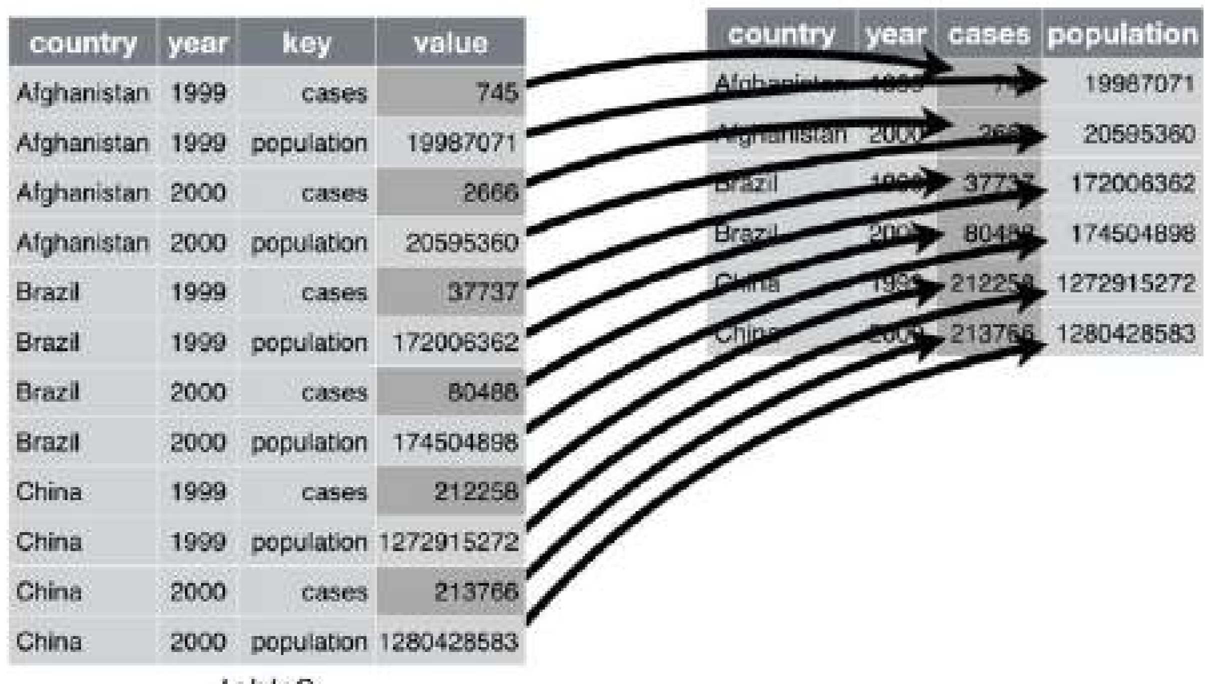
7. Use the gather() function as seen below.
dfPop2 = gather(dfPop, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`, key = 'YEAR', value = 'POPULATION')
8. View the output.
View(dfPop2)

	Country Name	Country Code	YEAR	POPULATION
1	Aruba	ABW	2010	101669
2	Afghanistan	AFG	2010	28803167
3	Angola	AGO	2010	23369131
4	Albania	ALB	2010	2913021
5	Andorra	AND	2010	84449
6	Arab World	ARB	2010	356508908
7	United Arab Emirates	ARE	2010	8270684
8	Argentina	ARG	2010	41223889
9	Armenia	ARM	2010	2877311
10	American Samoa	ASM	2010	55637
11	Antigua and Barbuda	ATG	2010	94661
12	Australia	AUS	2010	22031750
13	Austria	AUT	2010	8363404
14	Azerbaijan	AZE	2010	9054332

9. You can check your work against the solution file Chapter5_1.R.

Exercise 2: Spreading

Spreading is the opposite of gathering and is used when an observation is spread across multiple rows. In the diagram below, table2should define an observation of one country per year. However, you'll notice that this is spread across two rows. One row for cases and another for population.



country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

table2

We can use the spread()function to fix this problem. The spread() function takes two parameters: the column that contains variable names, known as the key and a column that contains values from multiple variables – the value.

```
spread(table2, key, value)
```

In this exercise you'll learn how to use the spread() function to resolve the types of problems we discussed in the introduction to this topic.

1. For this exercise you'll download some sample data that needs to be spread. Install the devtools package and DSR datasets using the code you see below by typing in the Console pane. Alternatively, you can use the **Packages** pane to install the packages.

```
install.packages("devtools")
devtools::install_github("garrettgman/DSR")
```

2. Load the DSR library by going to Package and clicking the check box next to DSR.
3. View table2. In this case, an observation is one country per year, but you'll notice that each observation is actually spread into two rows.

```
View(table2)
```

	country	year	type	count
1	Afghanistan	1999	cases	745
2	Afghanistan	1999	population	19987071
3	Afghanistan	2000	cases	2666
4	Afghanistan	2000	population	20595360
5	Brazil	1999	cases	37737
6	Brazil	1999	population	172006362
7	Brazil	2000	cases	80488
8	Brazil	2000	population	174504898
9	China	1999	cases	212258
10	China	1999	population	1272915272
11	China	2000	cases	213766
12	China	2000	population	1280428583

4. Use the spread() function to correct this problem.

```
table2b = spread(table2, key = type, value = count)
```

5. View the results.

```
View(table2b)
```

	country	year	cases	population
1	Afghanistan	1999	745	19987071
2	Afghanistan	2000	2666	20595360
3	Brazil	1999	37737	172006362
4	Brazil	2000	80488	174504898
5	China	1999	212258	1272915272
6	China	2000	213766	1280428583

6. You can check your work against the solution file Chapter5_2.R.

Exercise 3: Separating

Another common case involves two variables being placed into the same column. For example, the spreadsheet below has a State-County Name column that actually contains two variables separated by a slash.

1	STATE	STATEFIPS	COUNTYFIPS	FIPS	State-County Name
2	AL	1	1	1001	Alabama-Autauga County
3	AL	1	3	1003	Alabama-Baldwin County
4	AL	1	5	1005	Alabama-Barbour County
5	AL	1	7	1007	Alabama-Bibb County
6	AL	1	9	1009	Alabama-Blount County
7	AL	1	11	1011	Alabama-Bullock County
8	AL	1	13	1013	Alabama-Butler County
9	AL	1	15	1015	Alabama-Calhoun County
10	AL	1	17	1017	Alabama-Chambers County
11	AL	1	19	1019	Alabama-Cherokee County
12	AL	1	21	1021	Alabama-Chilton County
13	AL	1	23	1023	Alabama-Choctaw County
14	AL	1	25	1025	Alabama-Clarke County
15	AL	1	27	1027	Alabama-Clay County
16	AL	1	29	1029	Alabama-Cleburne County

The separate()function can be used to split a column into multiple columns by splitting on a separator. By default, the separate() function will automatically look for any nonalphanumeric character or you can define a specific character.

Here, the separate() function will split the values of the State-County Name column into two variables: StateAbbrev and CountyName.

The separate() function accepts parameters for the name of the column to separate along with the names of the columns to separate into, and an optional separator. By default, separate() will look for any non-alphanumeric character to use as the separator, but you can also define a specific separator. You can see an example of how the separate() function works below.

separate(table3, rate, into=c("cases", "population"))
In this exercise you'll learn how to use the separate() function to resolve the types of problems we discussed in the introduction to this topic.

1. In the Data folder where you installed the exercise data for this book is a file called usco2005.csv. Open this file, preferably in Microsoft Excel, or some other type of spreadsheet software. The file should look similar to the screenshot below.

STATE	STATEFIPS	COUNTYFIPS	FIPS	State-County Name	TP-TotPop
AL	1	1	1001	Alabama-Autauga County	48.612
AL	1	3	1003	Alabama-Baldwin County	162.586
AL	1	5	1005	Alabama-Barbour County	28.414
AL	1	7	1007	Alabama-Bibb County	21.516
AL	1	9	1009	Alabama-Blount County	55.725
AL	1	11	1011	Alabama-Bullock County	11.055
AL	1	13	1013	Alabama-Butler County	20.766
AL	1	15	1015	Alabama-Calhoun County	112.141
AL	1	17	1017	Alabama-Chambers County	35.46
AL	1	19	1019	Alabama-Cherokee County	24.522
AL	1	21	1021	Alabama-Chilton County	41.744
AL	1	23	1023	Alabama-Choctaw County	14.807
AL	1	25	1025	Alabama-Clarke County	27.269
AL	1	27	1027	Alabama-Clay County	13.964
AL	1	29	1029	Alabama-Cleburne County	14.46
AL	1	31	1031	Alabama-Coffee County	45.567
AL	1	33	1033	Alabama-Colbert County	54.66
AL	1	35	1035	Alabama-Conecuh County	13.257
AL	1	37	1037	Alabama-Coosa County	11.162
AL	1	39	1039	Alabama-Covington County	37.003
AL	1	41	1041	Alabama-Crenshaw County	13.727
AL	1	43	1043	Alabama-Cullman County	79.886
AL	1	45	1045	Alabama-Dale County	48.748
AL	1	47	1047	Alabama-Dallas County	44.366
AL	1	49	1049	Alabama-De Kalb County	67.271

2. Load the usco2005.csv file into RStudio by writing the code you see below in

the **Console** pane.
df = read_csv(“usco2005.csv”, col_names = TRUE)
3. View the imported data.
View(df)

	STATE	STATEFIPS	COUNTYFIPS	FIPS	State-County Name
1	AL	01	001	01001	Alabama-Autauga County
2	AL	01	003	01003	Alabama-Baldwin County
3	AL	01	005	01005	Alabama-Barbour County
4	AL	01	007	01007	Alabama-Bibb County
5	AL	01	009	01009	Alabama-Blount County
6	AL	01	011	01011	Alabama-Bullock County
7	AL	01	013	01013	Alabama-Butler County

4. Use the separate() function to separate the contents of the StateCounty Name column into StateAbbrev and CountyName columns.
df2 = separate(df,”State-County Name”,into = c(“StateAbbrev”, “CountyName”))
5. View the results.
View(df2)

	STATE	STATEFIPS	COUNTYFIPS	FIPS	StateAbbrev	CountyName	TP-TotPop
1	AL	1	1	1001	Alabama	Autauga	48.612
2	AL	1	3	1(1001	ibama	Baldwin	162.586
3	AL	1	5	1005	Alabama	Barbour	28.414
4	AL	1	7	1007	Alabama	Bibb	21.516
5	AL	1	9	1009	Alabama	Blount	55.725
6	AL	1	11	1011	Alabama	Bullock	11.055
7	AL	1	13	1013	Alabama	Butler	20.766

6. You can check your work against the solution file Chapter5_3.R.

Exercise 4: Uniting

The Unite()function is the exact opposite of separate() in that it combines multiple columns into a single column. While not used nearly as often as separate() , there may be times when you need the functionality provided by unite(). In this exercise you’ll unite the data frame that was separated in the last exercise.

1. In the **Console** pane, add the code you see below to unite the StateAbbrev and

CountyName columns back into a single column.
df3 = unite(df2, State_County_Name, StateAbbrev, CountyName)
2. View the result.
View(df3)

	STATE	STATEFIPS	COUNTYFIPS	FIPS	State_County_Name	TP-TotPop
1	AL	1	1	1001	Alabama_Autauga	48.612
2	AL	1	3	1003	Alabama_Baldwin	162.586
3	AL	1	5	1005	Alabama_Barbour	28.414
4	AL	1	7	1007	Alabama_Bibb	21.516
5	AL	1	9	1009	Alabama_Blount	55.725
6	AL	1	11	1011	Alabama_Bullock	11.055
7	AL	1	13	1013	Alabama_Butler	20.766
8	AL	1	15	1015	Alabama_Calhoun	112.141
9	AL	1	17	1017	Alabama_Chambers	35.460
10	AL	1	19	1019	Alabama_Cherokee	24.522
11	AL	1	21	1021	Alabama_Chilton	41.744
12	AL	1	23	1023	Alabama_Choctaw	14.807
13	AL	1	25	1025	Alabama_Clarke	27.269
14	AL	1	27	1025	Alabama_Clay	13.964

3. You can check your work against the solution file Chapter5_4.R.

Conclusion

In this chapter you were introduced to the tidyr package and its set of functions for creating tidy datasets. The next chapter will teach you the basics of data exploration using R and tidyverse.

Chapter 6

Basic Data Exploration Techniques in R

Exploratory Data Analysis (EDA) is a workflow designed to gain a better understanding of your data. The workflow consists of three steps. The first is to generate questions about your data. In this step you want to be as broad as possible because at this point you don't really have a good feel for the data. Next, search for answers to these questions by visualizing, transforming, and modeling the data. Finally, refine your questions and or generate new questions. In R there are two primary tools that support the data exploration process: plots and summary statistics.

Data can generally be divided into categorical or continuous types. Categorical variables consist of a small set of values, while continuous variables have a potentially infinite set of ordered values. Categorical variables are often visualized with bar charts, and continuous variables with histograms. Both categorical and continuous data can be represented through various charts created with R.

When performing basic visualization of variables, we tend to measure either variation or covariation. Variation is the tendency of the values of a variable to change from measurement to measurement. The variable being measured is the same though. This would include things like the total acres burned by a wildfire (continuous) or the number of crimes by police district (categorical data. Covariation is the tendency of the values of two or more variables to vary together in a related way.

- Measuring categorical variation with a bar chart
- Measuring continuous variation with a histogram
- Measuring covariation with boxplots
- Measuring covariation with symbol size
- Creating 2D bins and hex charts
- Generating summary statistics

Exercise 1: Measuring Categorical Variation with a Bar Chart

A bar chart is a great way to visualize categorical data. It separates each category into a separate bar and then the height of each bar is defined by the number of

occurrences in that category.

1. The exercises in this chapter require the following packages: readr, dplyr, ggplot2. They can be loaded from the **Packages** pane, the **Console** pane, or a script.

2. Open RStudio and find the **Console** pane.

3. If necessary, set the working directory by typing the code you see below into the **Console** pane or by going to **Session | Set Working Directory | Choose Directory** from the RStudio menu.

```
setwd(<installation directory for exercise data>)
```

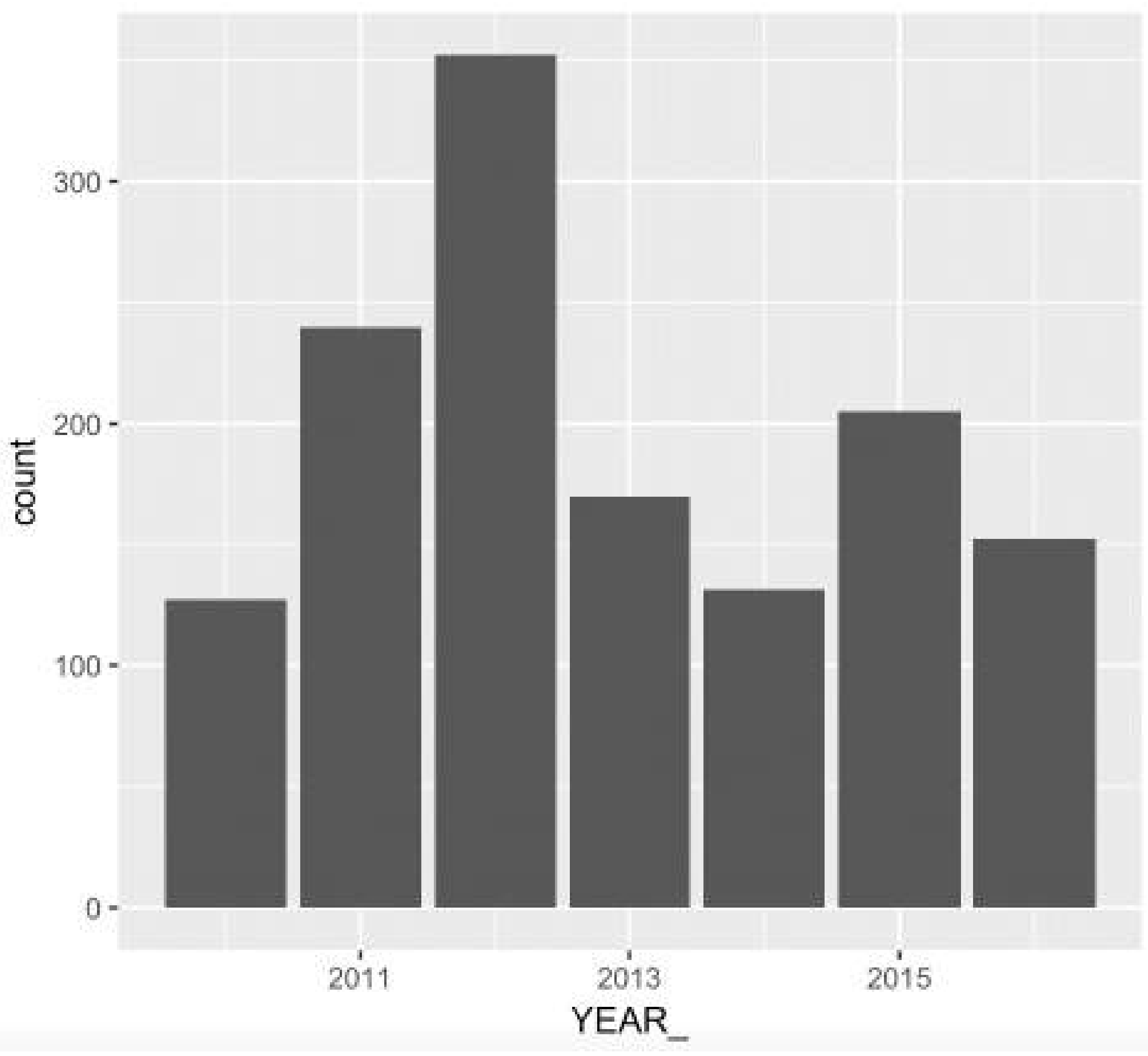
4. Use the read_csv()
function to load the dataset into a data frame.

```
dfFires <- read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
```

5. For this analysis, we'll filter the data so that only fires that burned greater than 1,000 acres in the years 2010 through 2016 are represented. Add the code you see below to filter the data and and send the results to a bar chart.

```
df <- filter(df, TOTALACRES >= 1000, YEAR_ %in% c(2010, 2011, 2012, 2013, 2014, 2015, 2016))
ggplot(data = df) + geom_bar(mapping = aes(x = YEAR_))
```

This will produce a bar chart that appears as seen in the screenshot below.



6. Use the count() function to get the actual count for each category.
View(count(df, YEAR_))

31	2010	9115
32	2011	10400
33	2012	10857
34	2013	11231
35	2014	9743
36	2015	9950
37	2016	7204

Exercise 2: Measuring Continuous Variation with a Histogram

The distribution of a continuous variable can be measured with the use of a histogram. In this exercise you'll create a histogram of wildfire acres burned.
1. On a new line, use the read_csv() function to load the StudyArea.csv file.

```
dfFires <- read_csv("StudyArea.csv", col_types = list(UNIT = col_character()),
col_names = TRUE)
```

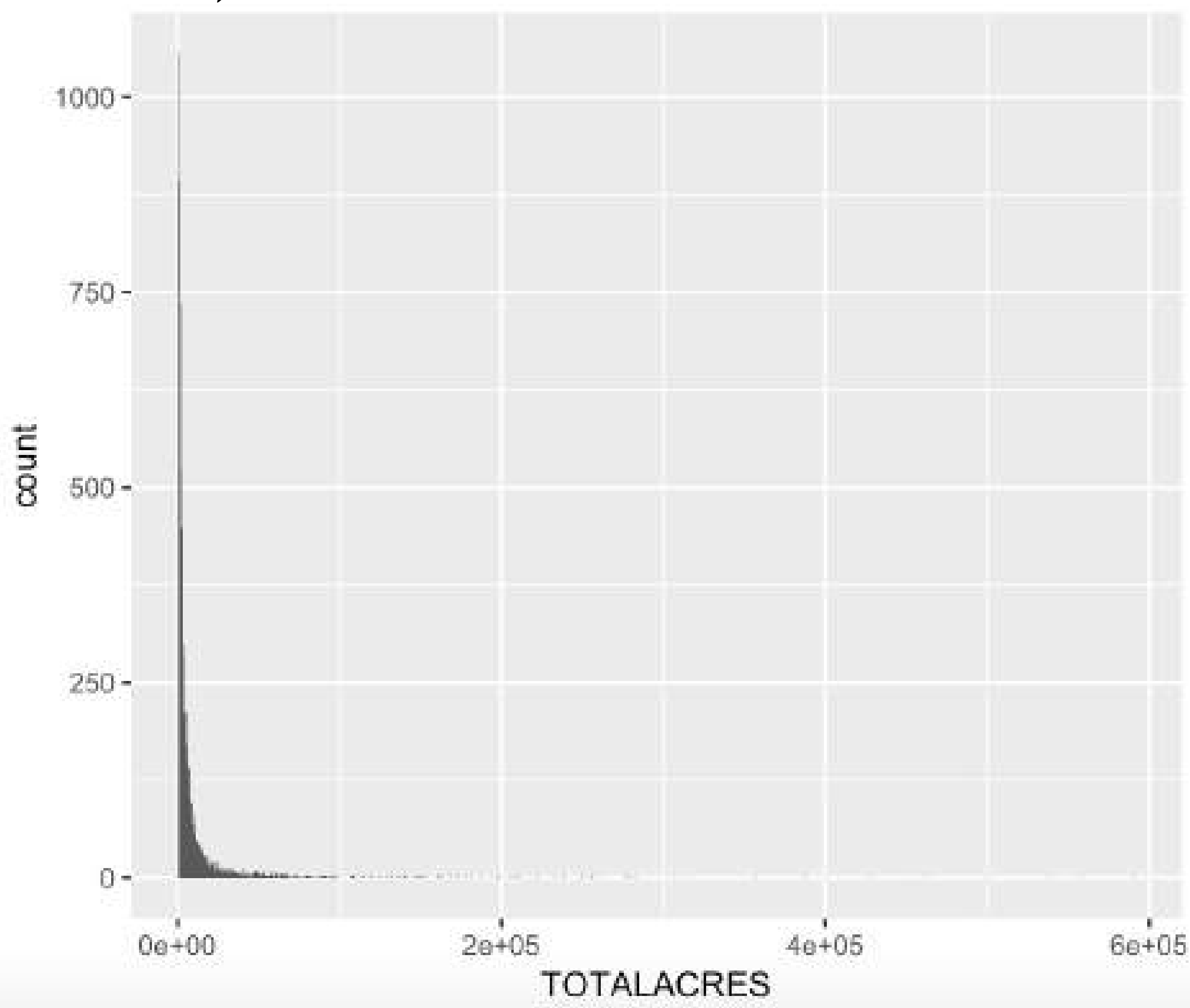
2. Pipe the data frame and use the select() function to limit the columns and filter the rows so that only fires greater than 1,000 acres are included. Since we have a large number of wildfires that burned only a small number of acres we'll focus on fires that are a little larger in this case.

```
df %>%
select(ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE) %>%
filter(TOTALACRES >= 1000) %>%
```

3. Create the histogram using ggplot() with geom_hist() and a bin size of 500. The data is obviously still skewed toward the lower end of the number of acres burned. Add the highlighted code you see below to produce the chart.

```
df %>%
select(ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE) %>%
filter(TOTALACRES >= 1000) %>%
```

ggplot() + geom_histogram(mapping = aes(x=TOTALACRES), binwidth=500)



4. You can also get a physical count of the number of fires that fell into each bin. From viewing the histogram and the count it's obvious that the vast majority of fires are small.

```
df %>%
count(cut_width(TOTALACRES, 500))

`cut_width(TOTALACRES, 500)` n
<fct> <int>
1 [750,1250] 154
2 (1250,1750] 178
3 (1750,2250] 144
```

4 (2250,2750] 82
5 (2750,3250] 70
6 (3250,3750] 39
7 (3750,4250] 59
8 (4250,4750] 42
9 (4750,5250] 40
10 (5250,5750] 37

5. Challenge: Recreate the histogram using a bin size of 5000. What is the effect on the output?

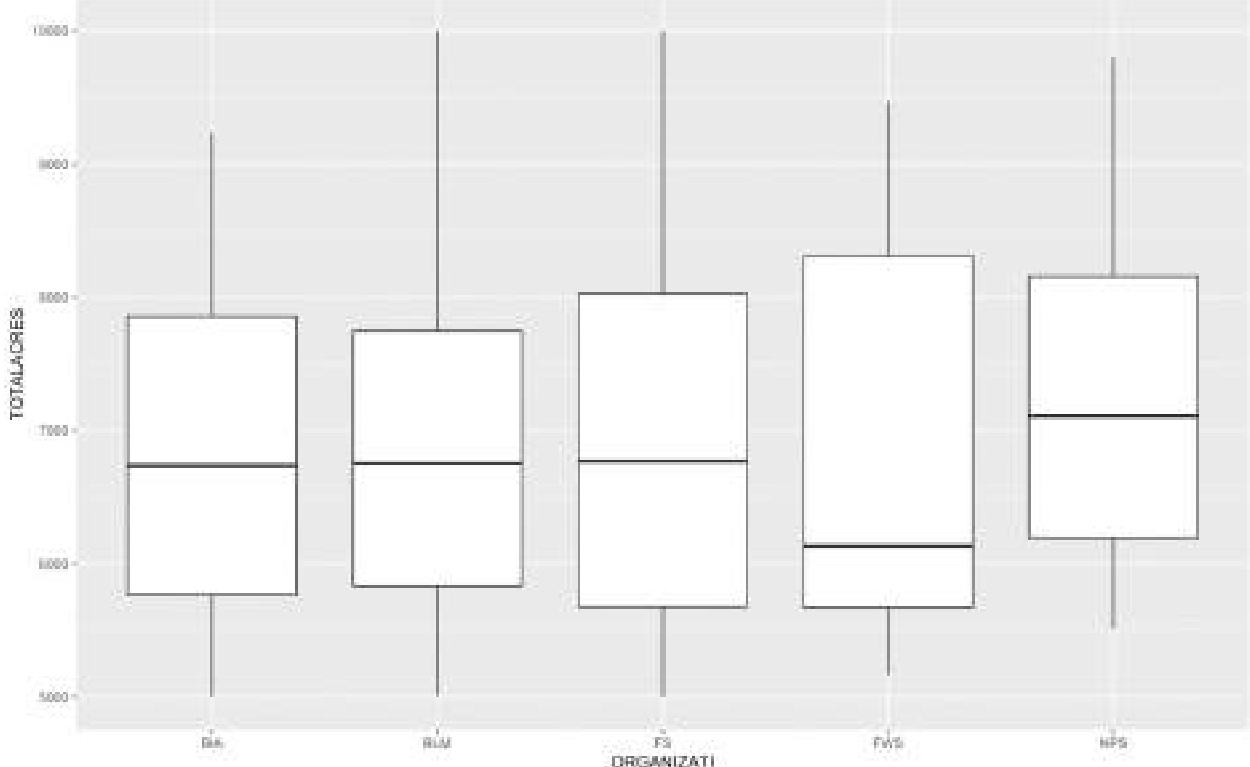
Exercise 3: Measuring Covariation with Box Plots

Box plots provide a visual representation of the spread of data for a variable. These plots display the range of values for a variable along with the median and quartiles. Follow the instructions provided below to create a box plot that measures covariation between organization and total acreage burned.

1. Use the
read_csv() function to load the dataset into a data frame.
dfFires <- read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)

2. Pipe the data frame and filter the rows so that only fire between 5000 and 1000 acres are included. Then, group the data by organization. The ORGANIZATI column in the dataset contains categorical data for the U.S. federal government agencies that have had land affected by wildfires. Finally, use ggplot() with geom_boxplot() to create a boxplot showing the distribution of wildfires by organization.

```
df %>%
filter(TOTALACRES >= 5000 & TOTALACRES <= 10000) %>%
group_by(ORGANIZATI) %>%
ggplot(mapping = aes(x = ORGANIZATI, y = TOTALACRES)) + geom_
boxplot()
```



The organization is listed on the X axis and the total acreage burned on the Y axis. The box contains a horizontal line that represents the median for the variable and the box itself is known as the Inter Quartile Range (IQR). The vertical lines that extend on either side of the box are known as the whiskers and represent the first and fourth quartile. A larger box and whiskers indicate a larger distribution of data.

3. Challenge: Create a new boxplot that maps the covariation of CAUSE and TOTALACRES.

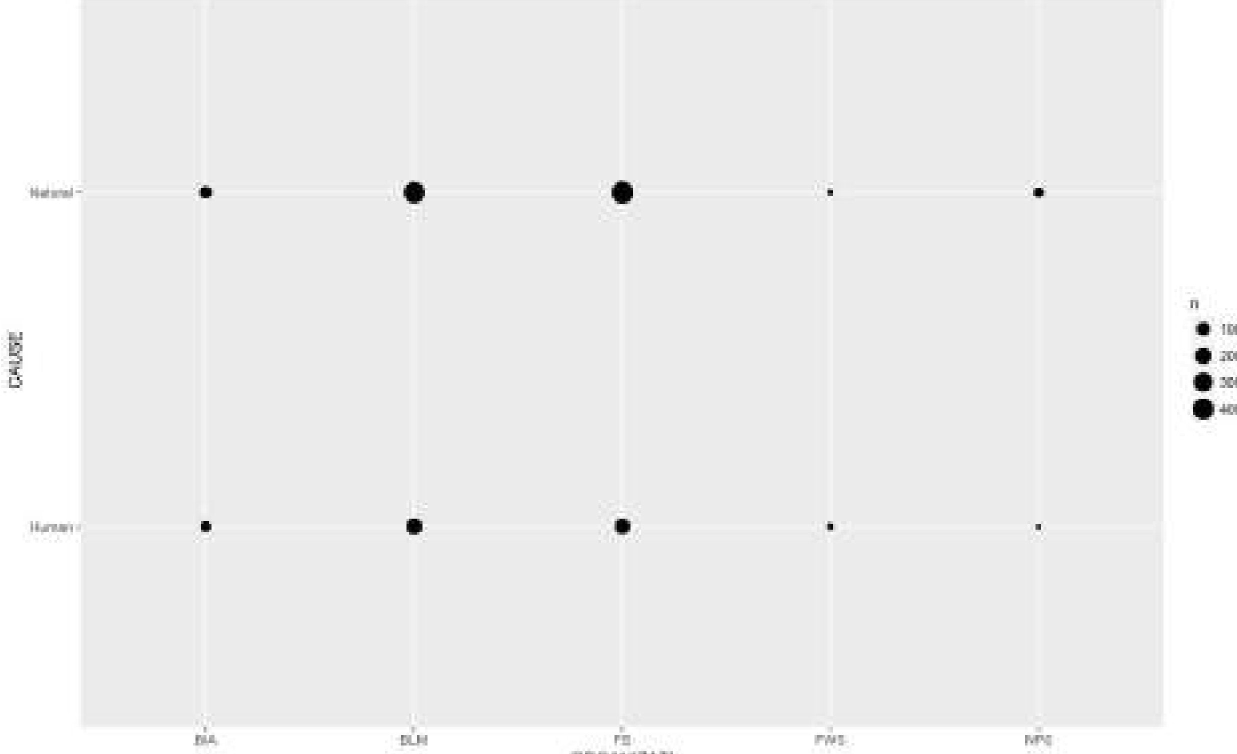
Exercise 4: Measuring Covariation with Symbol Size

The geom_count() function can be used with ggplot() to measure covariation between variables using different symbol sizes. Follow the instructions provided below to measure the covariation between organization and wildfire cause using symbol size.

1. Use the read_csv() function to load the dataset into a data frame.
dfFires <- read_csv(“StudyArea.csv”, col_types = list(UNIT = col_character()), col_names = TRUE)

2. Pipe the data frame and filter the rows so that only wildfires that originated due to Natural or Human causes are included. This will remove any records that are Unknown or have missing values. Then, use geom_count() to create a graduated symbol chart based on the number of fires by organization.

```
df %>%
filter(CAUSE == ‘Natural’ | CAUSE == ‘Human’) %>%
group_by(ORGANIZATI) %>%
ggplot() + geom_count(mapping = aes(x = ORGANIZATI, y = CAUSE))
```



3. You can also get an exact count of the number of fires by organization and cause.

```
df %>%
count(ORGANIZATI, CAUSE)
```

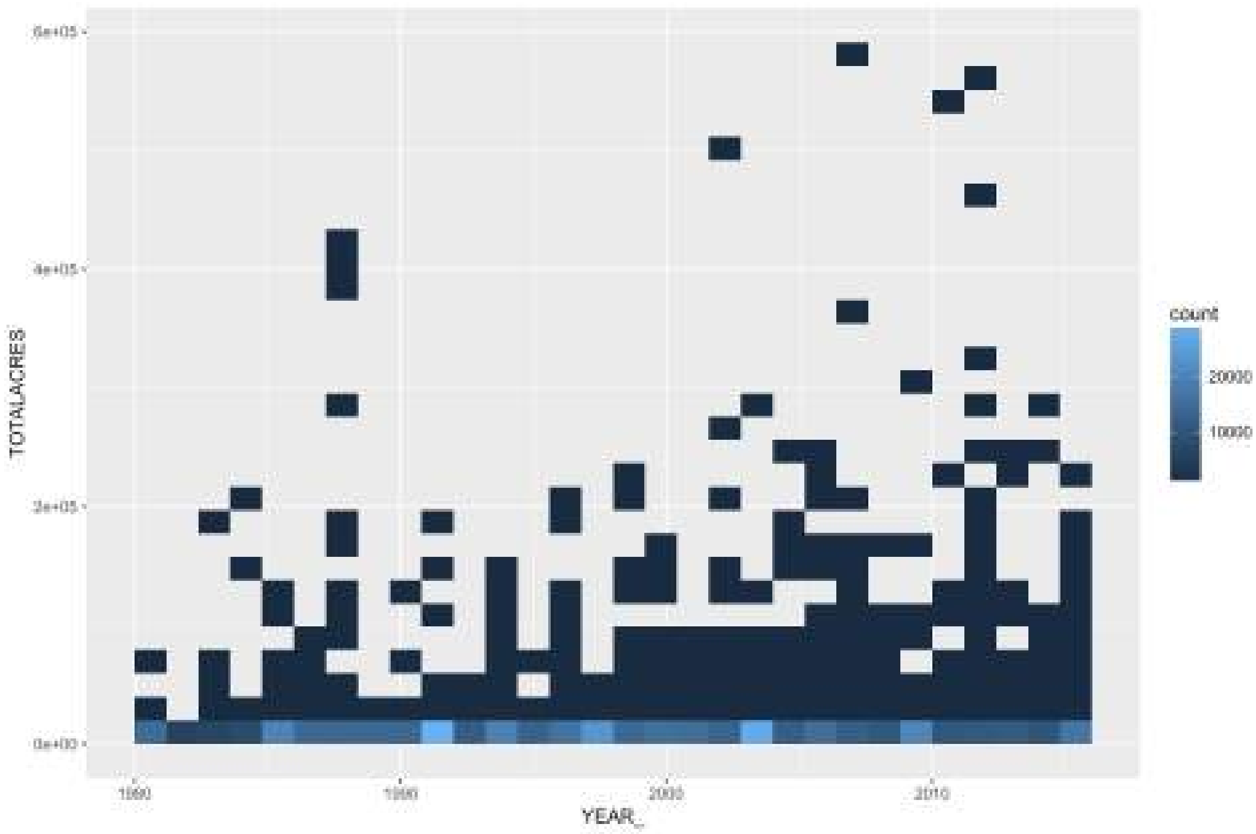
```
ORGANIZATI CAUSE n
<chr> <chr> <int>
1 BIA Human 49
2 BIA Natural 91
3 BLM Human 187
4 BLM Natural 386
5 FS Human 158
```

6 FS Natural 431
7 FWS Human 10
8 FWS Natural 7
9 FWS Undetermined 6
10 NPS Human 6
11 NPS Natural 46

Exercise 5: 2D bin and hex charts

You can also use 2D bin and hex charts as an alternative way of viewing the distribution of two variables. Follow the instructions provided below to create 2D bin and hex charts that visualize the relationship between the year and total acreage burned.

1. Use the
read_csv() function to load the dataset into a data frame.
dfFires <- read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
2. Create a 2D bin map with YEAR_ on the X axis and TOTALACRES on the Y axis.
ggplot(data = dfFires) + geom_bin2d(mapping = aes(x=YEAR_, y=TOTALACRES))



3. Create a 2D hex map with YEAR_ on the X axis and TOTALACRES on the Y axis.

ggplot(data=df) + geom_hex(mapping = aes(x=YEAR_, y=TOTALACRES))

Exercise 6: Generating Summary Statistics

Another basic technique for performing exploratory data analysis is to generate various summary statistics on a dataset. R includes a number of individual functions for generating specific summary statistics or you can use the summary() function to generate a set of summary statistics.

1. Reload the StudyArea.csv file into a data frame.
df <- read_csv(“StudyArea.csv”, col_types = list(UNIT = col_character()), col_names = TRUE)
2. Restrict the list of columns.
df <- select(df, ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE)
3. Filter the list to include only wildfires greater than 1,000 acres.
df <- filter(df, TOTALACRES >= 1000)
4. Call the mean() function, passing in a reference to the data frame and the TOTALACRES column.
mean(df$TOTALACRES)
[1] 10813.06
5. Call the median() function.
median(df$TOTALACRES) [1] 3240
6. Instead of calling the individual summary statistics functions you can simply use the summary() function to return a list of summary statistics.
summary(df$TOTALACRES)
Min. 1st Qu. Median Mean 3rd Qu. Max. 1000 1670 3240 10813 8282 590620
7. You can check your work against the solution file Chapter6_6.R.

Conclusion

In this chapter you learned some basic data exploration techniques using R. You learned how to measure categorical and continuous variation with bar charts and histograms, and covariation with box plots and different symbol size. Finally, you learned how to generate summary statistics and create 2D bins and hex charts.

In the next chapter you’ll learn how to visualize data using the ggplot2 package.
Chapter 7

Basic Data Visualization Techniques

The ggplot2 package is a library that enables the creation of many types of data visualization including various types of charts and graphs. This library was first created by Hadley Wickham in 2005 and is an R implementation of Leland Wilkinson's Grammar of Graphics. The idea behind this package is to specify plot building blocks and then combine them to create a graphical display. Building blocks of ggplot2 include data, aesthetic mapping, geometric objects, statistical transformations, scales, coordinate systems, position adjustments, and faceting.

There are a number of advantages to using ggplot2versus other visualization techniques available in R. These advantages include a consistent style for defining the graphics, a high level of abstraction for specifying plots, flexibility, a built-in theming system for plot appearance, mature and complete graphics system, and access to many other ggplot2 users for support.

In this chapter we'll cover the following topics:
- Creating a scatterplot
- Adding a regression line to a scatterplot
- Plotting categories
- Labeling the graph
- Legend layouts
- Creating a facet
- Theming
- Creating bar charts
- Creating violin plots
- Creating density plots

Step 1: Creating a scatterplot

A scatterplot is a graph in which the values of two variables are plotted along two axes, with the pattern of the resulting points revealing any correlation present.

1. The exercises in this chapter require the following packages: readr, dplyr, ggplot2. They can be loaded from the **Packages** pane, the **Console** pane, or a

script.

2. Open RStudio and find the **Console** pane.

3. If necessary, set the working directory by typing the code you see below into the **Console** pane or by going to **Session | Set Working Directory | Choose Directory** from the RStudio menu.

```
setwd(<installation directory for exercise data>)
```

4. Load the contents of the StudyArea.csv file into a data frame.

```
dfWildfires <- read_csv(“StudyArea.csv”, col_types = list(UNIT = col_character()), col_names = TRUE)
```

5. Create a subset of columns.

```
df <- select(dfWildfires, ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE)
```

6. Group the records by year.

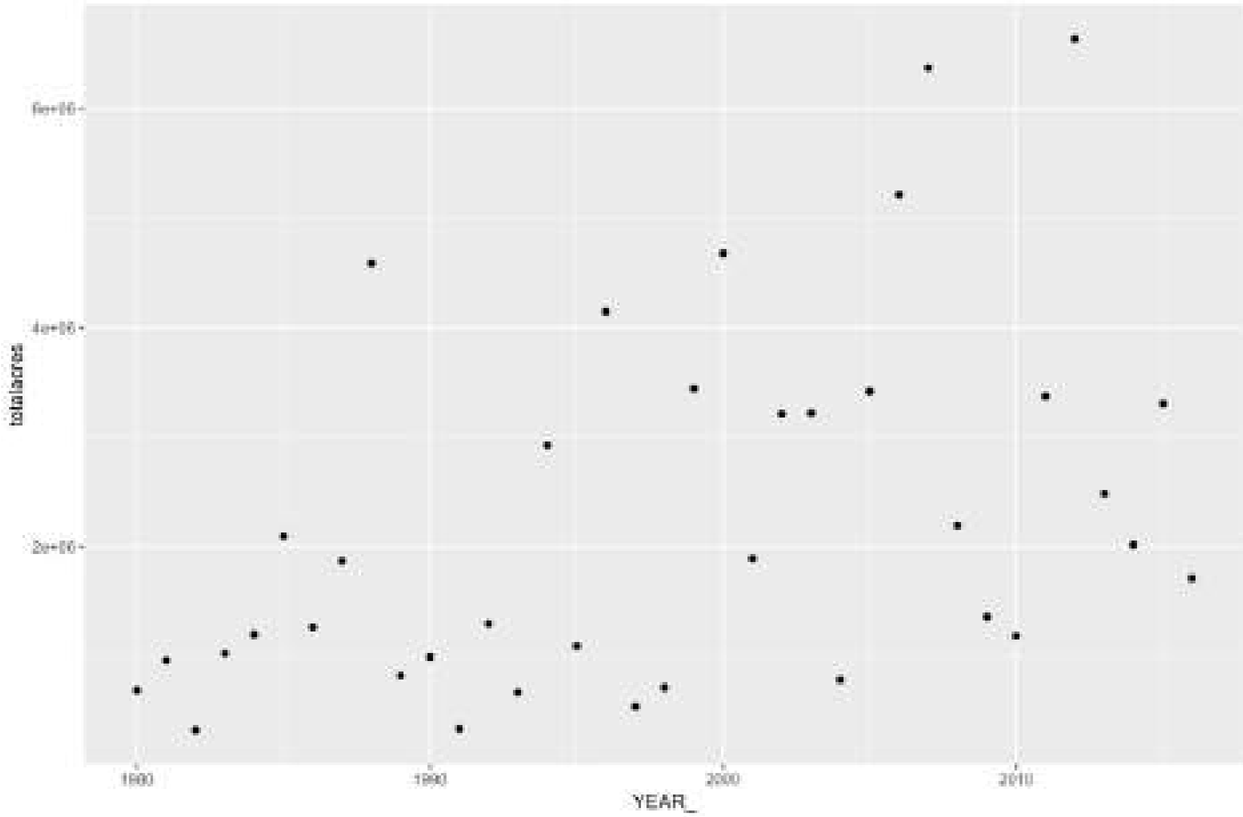
```
grp <- group_by(df, YEAR_)
```

7. Summarize the data by total number of acres burned.

```
sm <- summarize(grp, totalacres = sum(TOTALACRES))
```

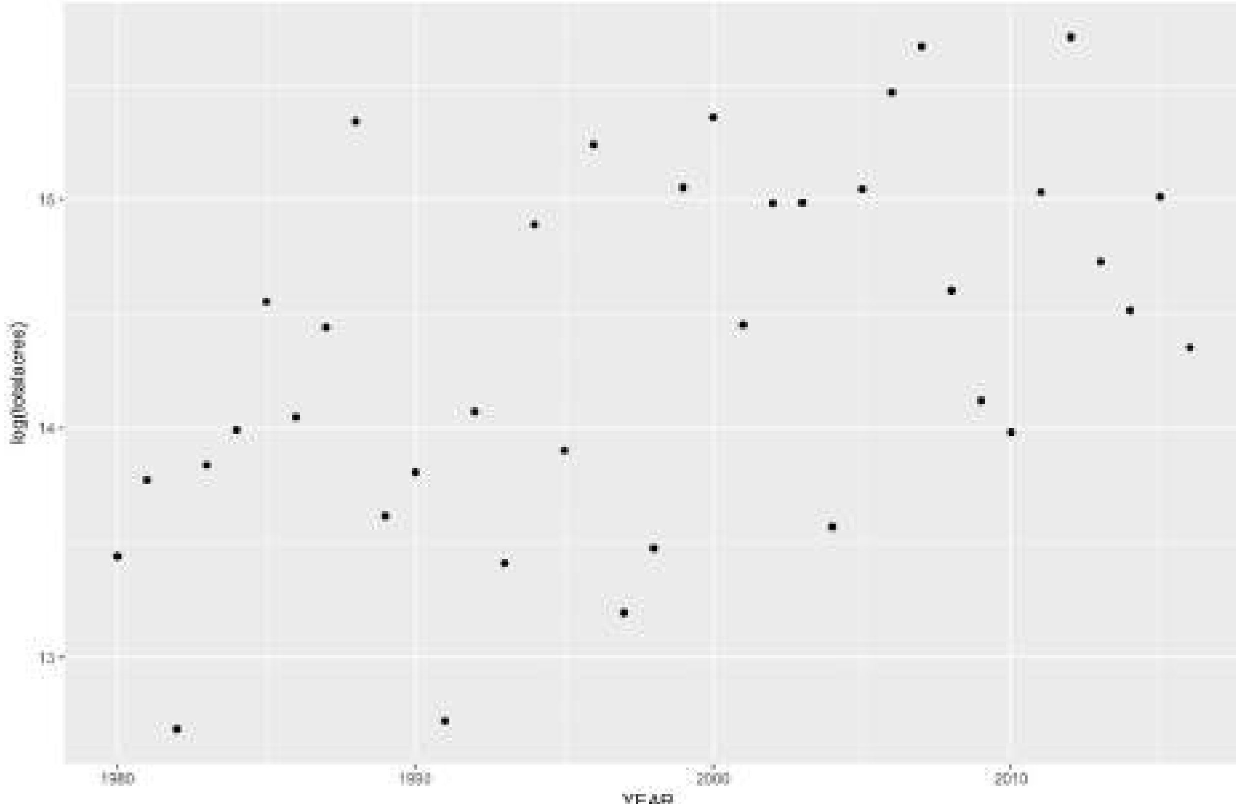
8. Use ggplot() to create a scatterplot with the year on the x axis and the total acres burned on the y axis.

```
ggplot(data=sm) + geom_point(mapping = aes(x=YEAR_, y=totalacres))
```



9. There are times when it makes sense to use the logarithmic scales in charts and graphs. One reason is to respond to skewness towards large values, i.e, cases in which one or a few points are much larger than the bulk of the data. In the graph that we just created there are a couple points that fall into this category on the y axis.

Create the graph again, but this time use the
log() function on the totalacres column.
ggplot(data=sm) + geom_point(mapping = aes(x=YEAR_, y=log(totalacres)))



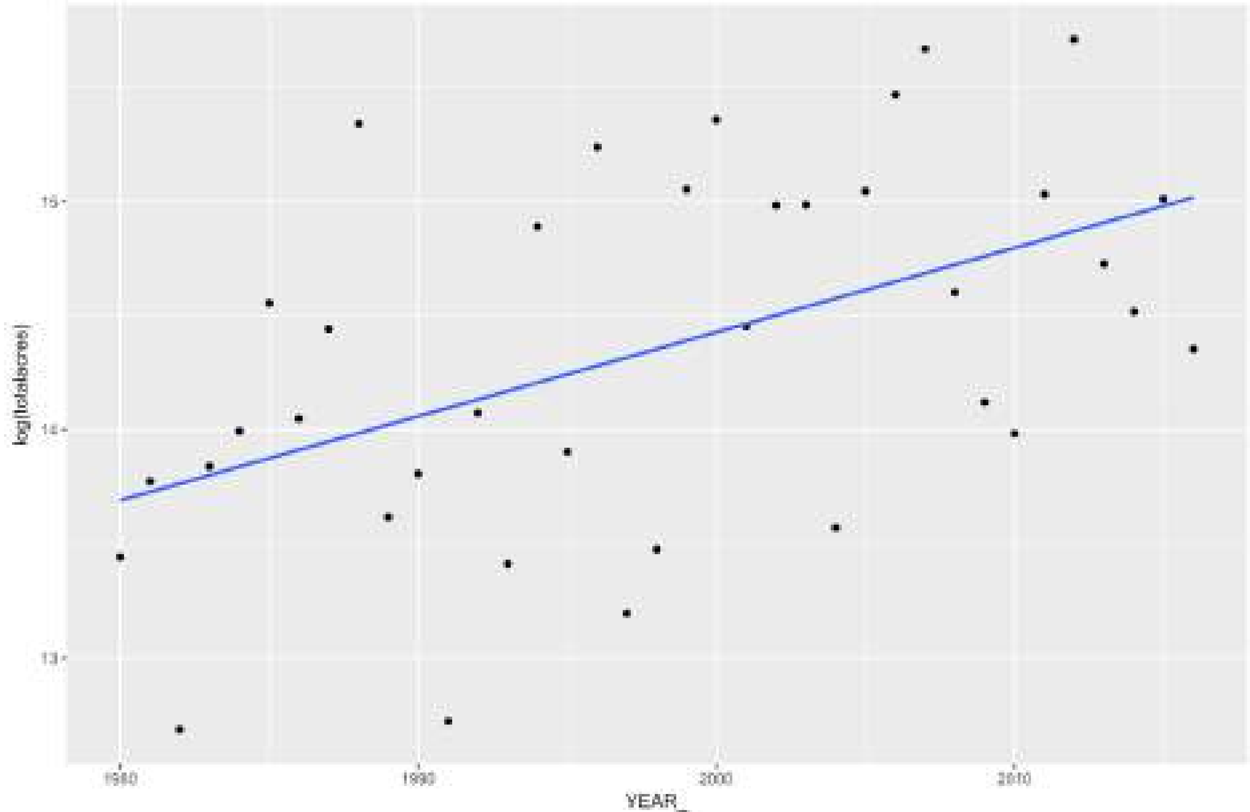
10. You can check your work against the solution file Chapter7_1.R.

Step 2: Adding a regression line to the scatterplot

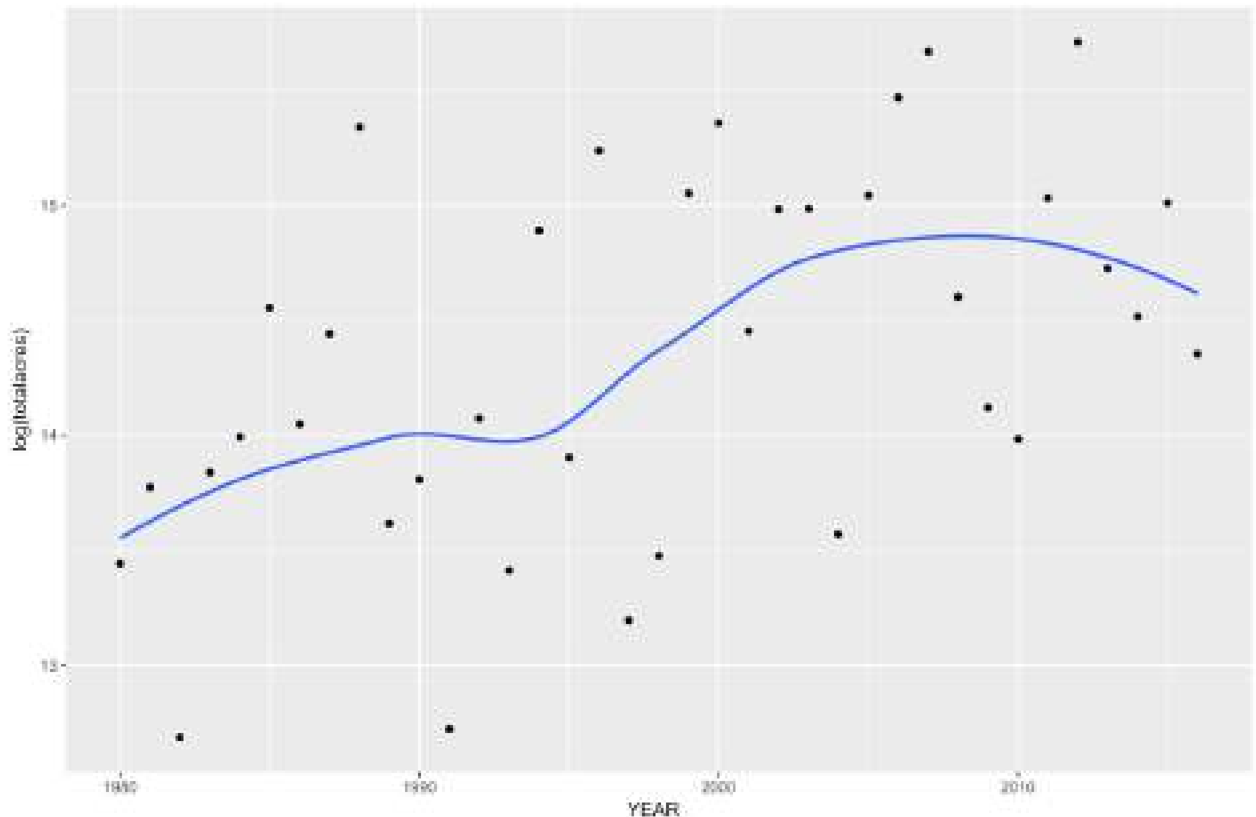
Plots constructed with ggplot() can have more than one geometry. It's common to add a prediction (regression) line to the plot.

1. There are several ways that you can add a regression line to the scatterplot, one of which is to use the geom_smooth() function with the method set to lm (straight line) and the se parameter set to FALSE. Add the line of code you see below to the console window.

```
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() +
geom_smooth(method=lm, se=FALSE)
```

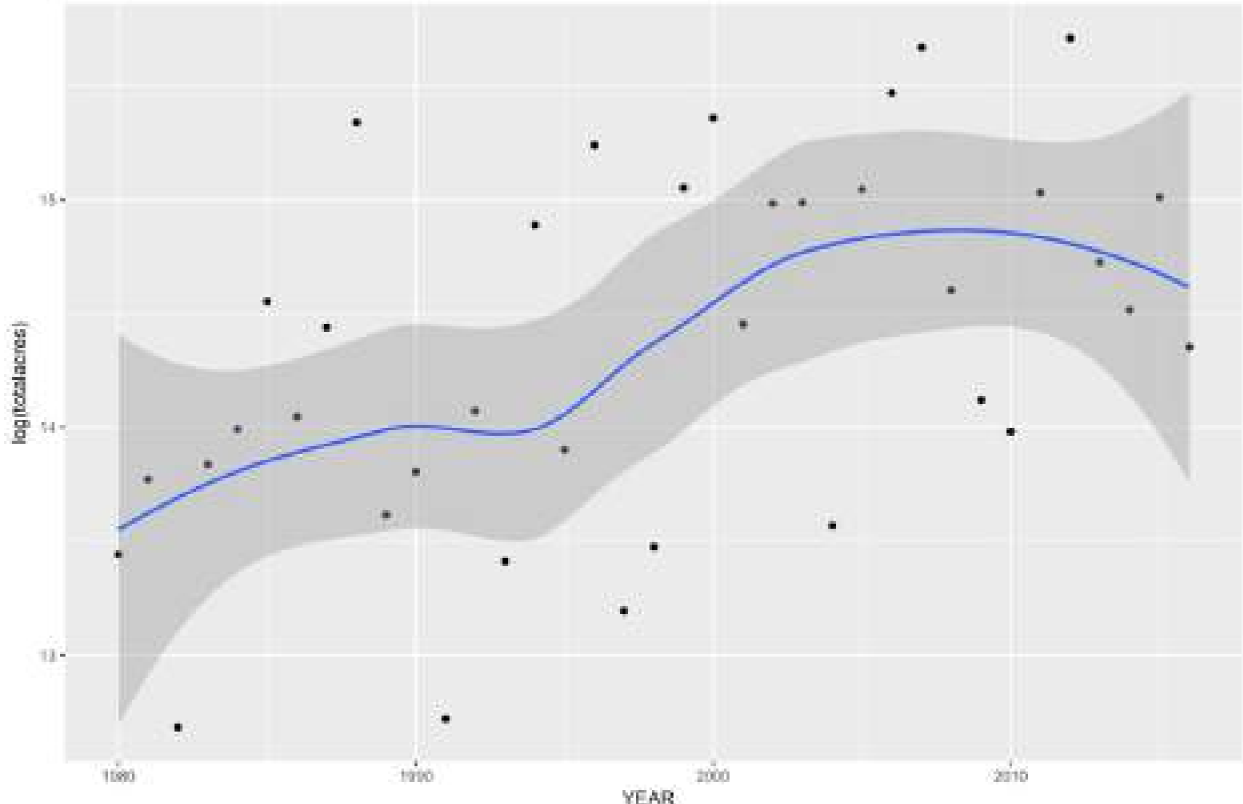


2. Change the method to loess the effect on the regression line.
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() +
geom_smooth(method=loess, se=FALSE)



3. You can add a confidence interval around the regression line by setting se = TRUE.

ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() + geom_smooth(method=loess, se=TRUE)

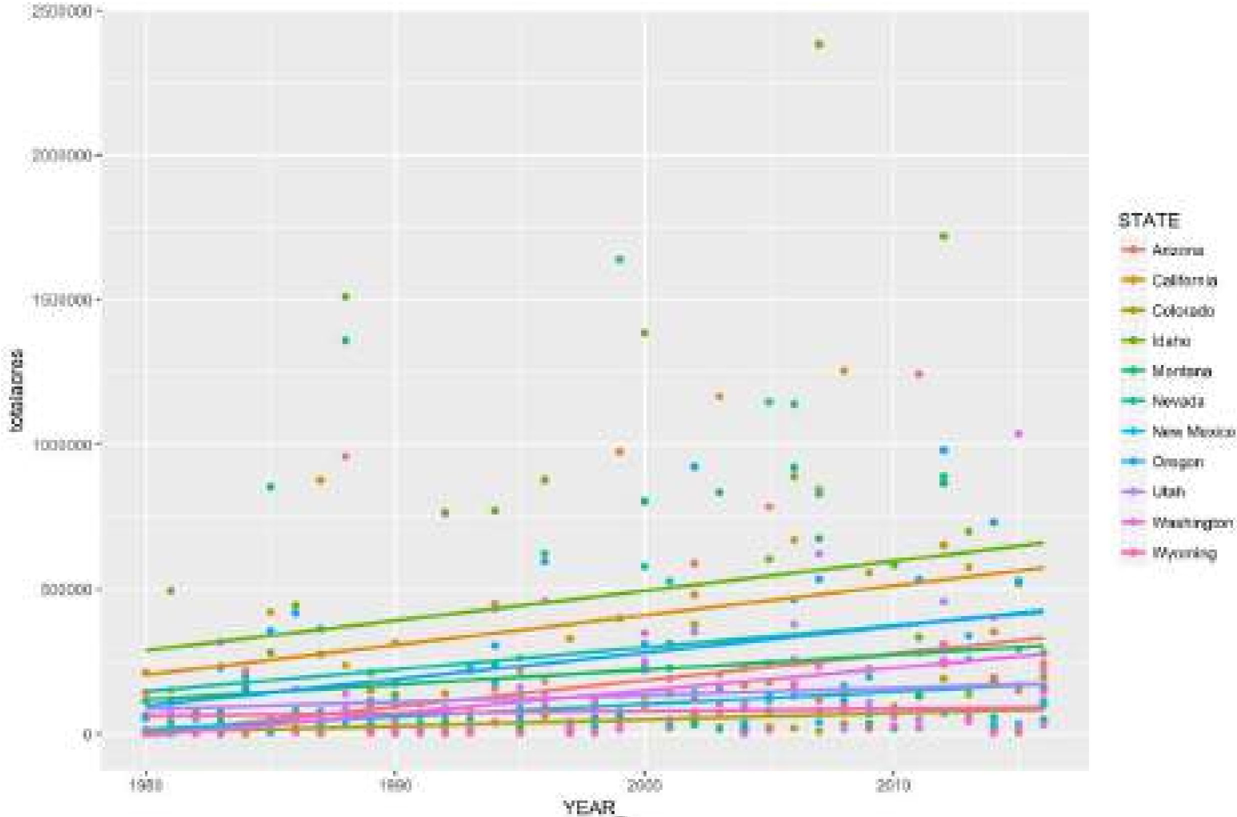


4. You can check your work against the solution file Chapter7_2.R.

Step 3: Plotting categories

Rather than graphing the entire set of wildfires you might want to better understand the trends by state. In this step you'll create a new scatterplot that visualizes wildfires trends over time by state.

1. Regroup the wildfires data frame by state and year.
grp <- group_by(df, STATE, YEAR_)
2. Summarize the groups by total acres burned.
sm <- summarize(grp, totalacres = sum(TOTALACRES))
3. Add a
colour parameter to the aes() function so that the points and regression line are mapped according to the state in which they occurred.
ggplot(data=sm, aes(x=YEAR_, y=totalacres, colour=STATE)) + geom_
point(aes(colour = STATE)) + stat_smooth(method=lm, se=FALSE)



4. You can check your work against the solution file Chapter7_3.R

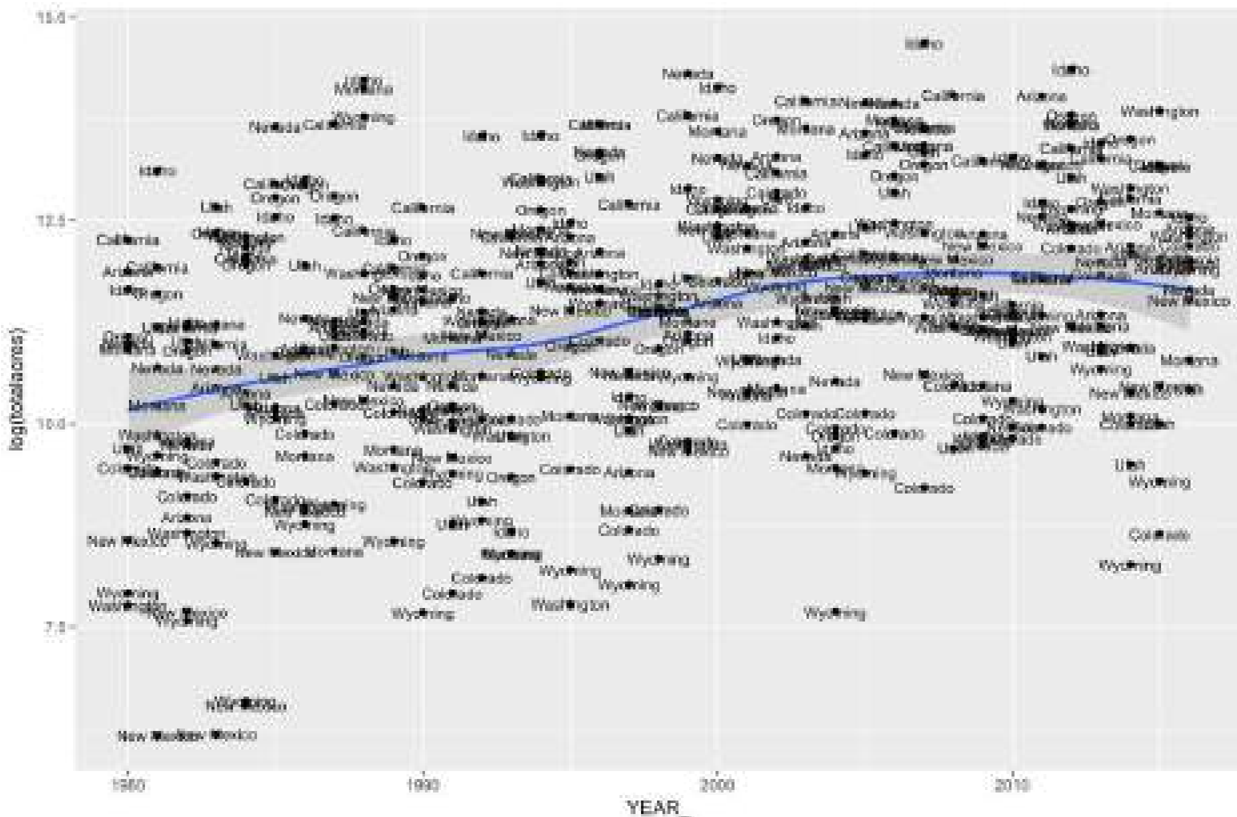
Step 4: Labeling the graph

You can add labels to your graph through either the geom_text() function or the geom_ label() function.

1. Label each of the points on the scatterplot using geom_text() with a label size of 3

.

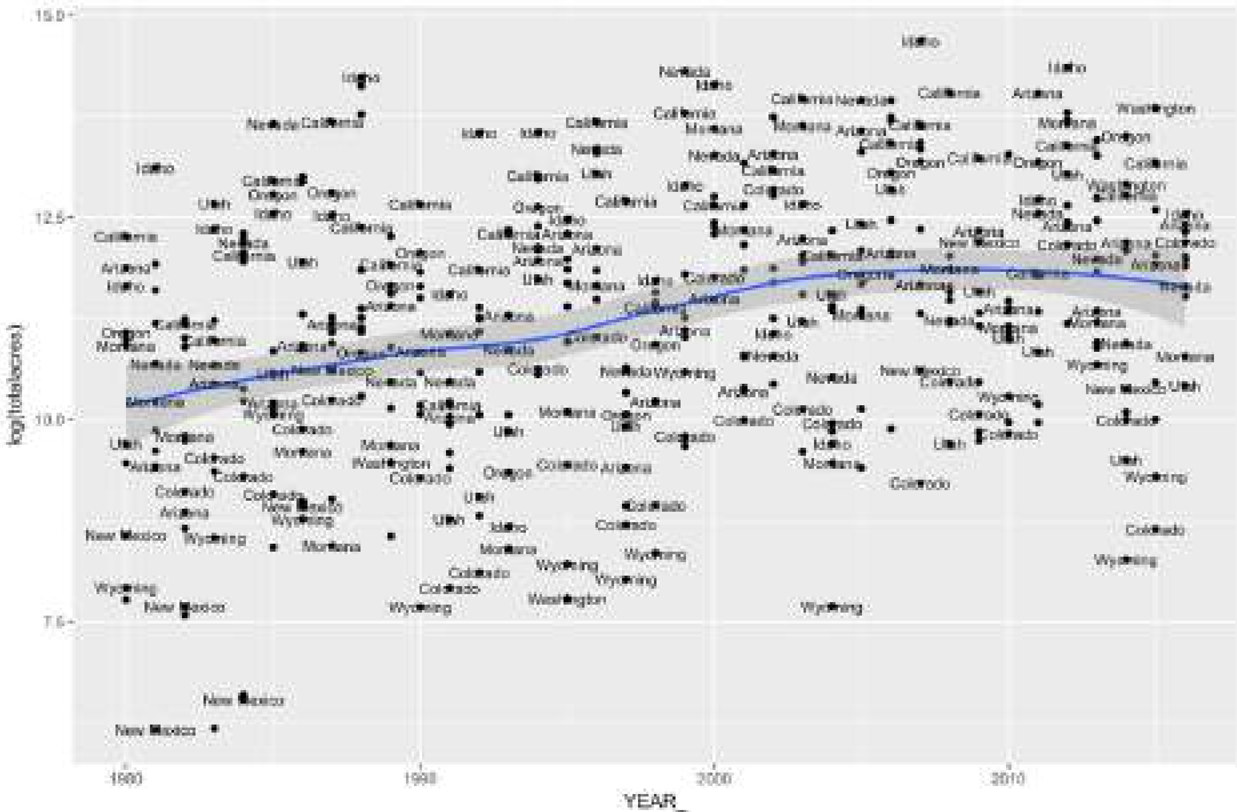
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() + geom_smooth(method=loess, se=TRUE) + geom_text(aes(label=STATE), size=3)



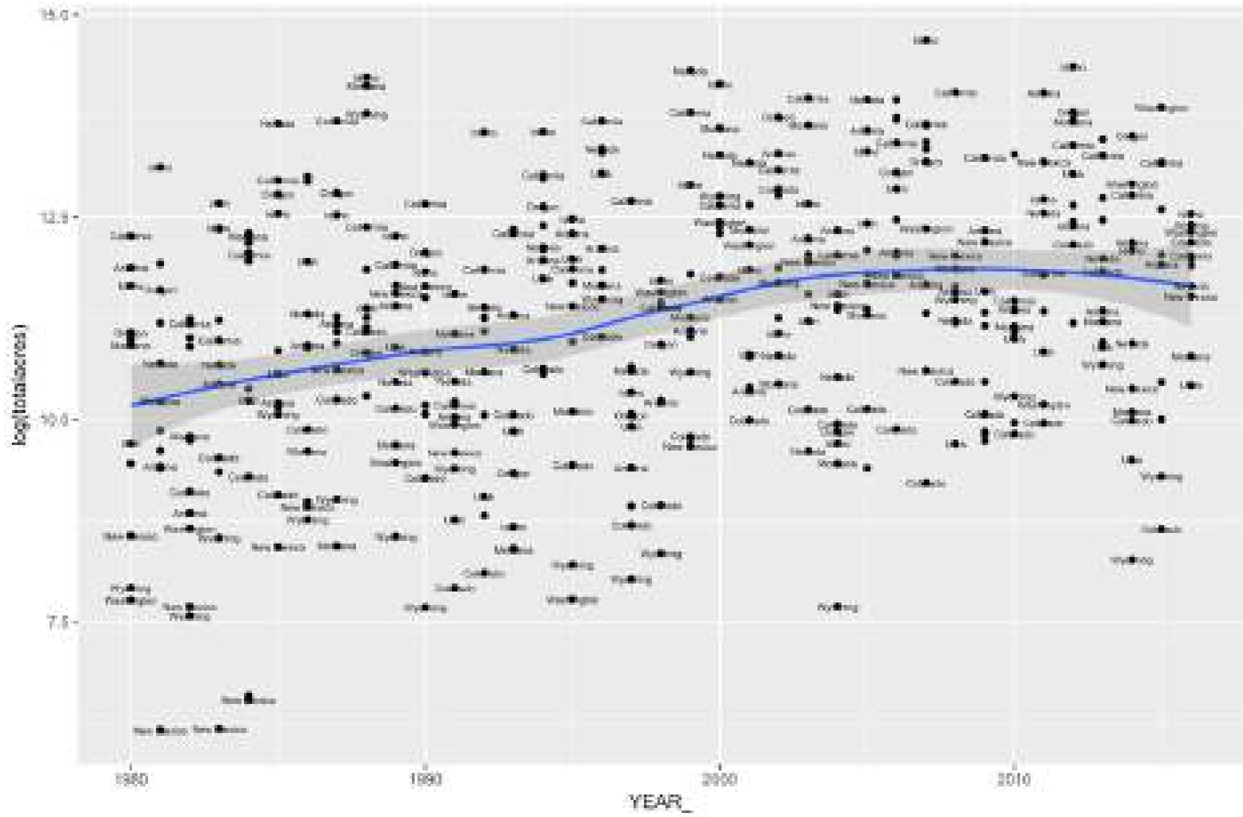
Now this obviously doesn’t work very well. The display is extremely cluttered so let’s adjust a few parameters to make this easier to read.

2. You can use the check_overlap parameter to remove any overlapping labels. Update your code as seen below.

```
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() +
geom_smooth(method=loess, se=TRUE) + geom_text(aes(label=STATE),
size=3, check_overlap = TRUE)
```

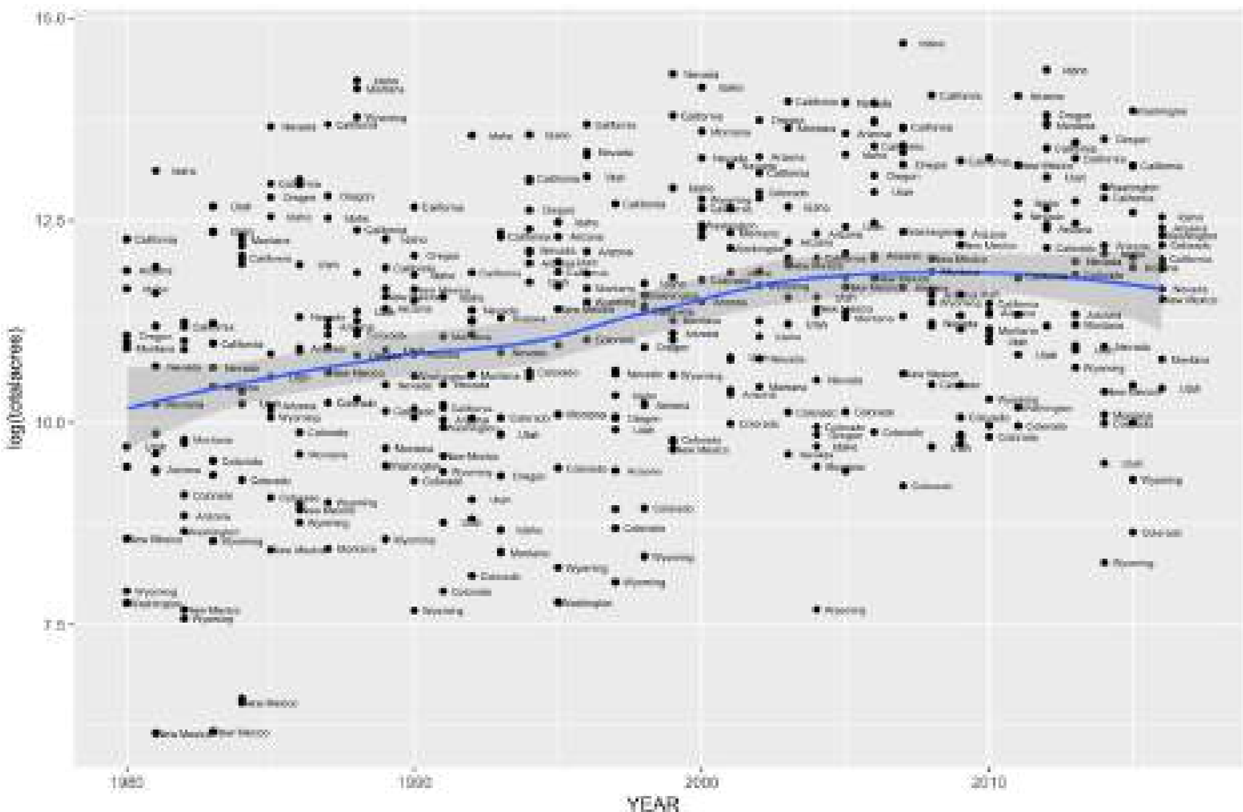


3. This look quite a bit better but if you change the label size to 2 it will further reduce the clutter and overlapping while hopefully still being readable.



4. You may have noticed that the labels sit directly on top of the topics. You can use the nudge_x and nudge_y parameters to move the labels relative to the point. Use nudge_x as seen below to see how this moves the labels horizontally.

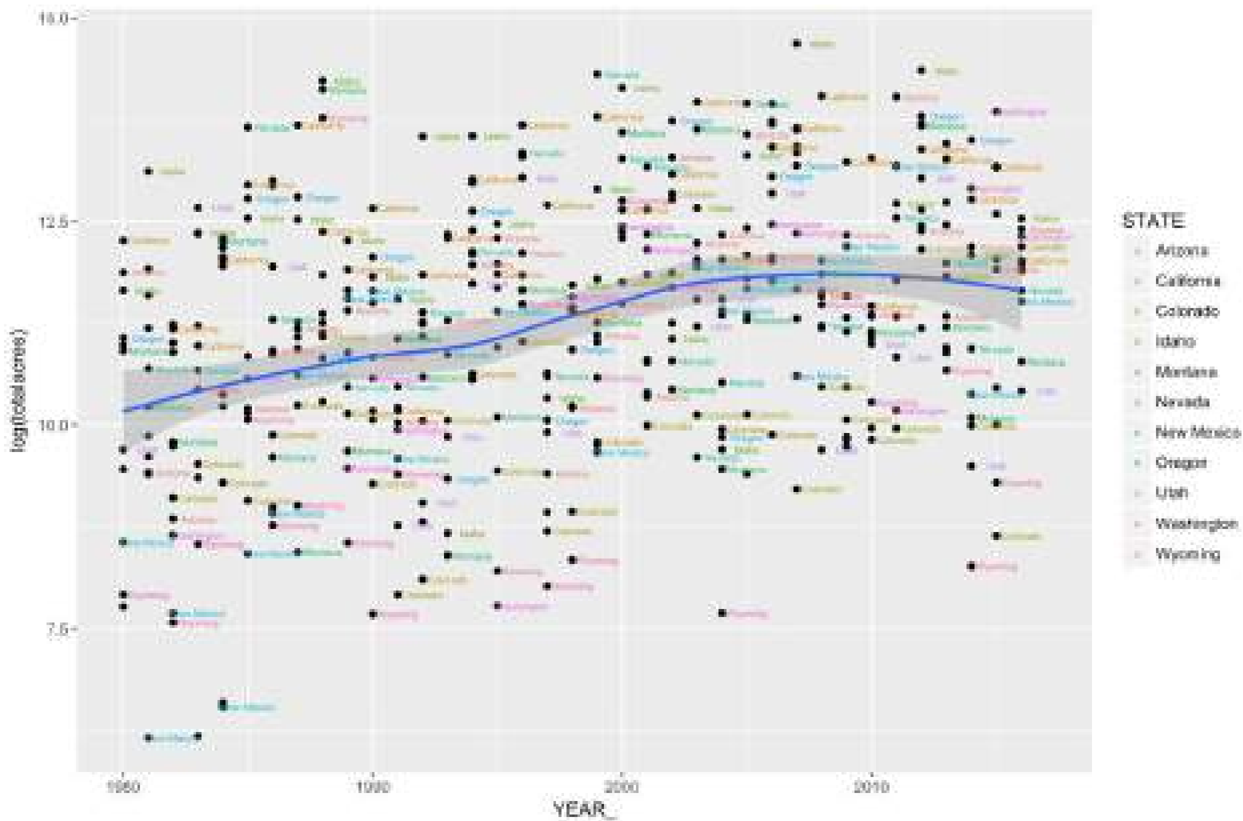
```
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() +
geom_smooth(method=loess, se=TRUE) + geom_text(aes(label=STATE),
size=2, check_overlap = TRUE, nudge_x = 1.0)
```



5. You can also color the labels by category by adding the color parameter to the aes() for geom_text()

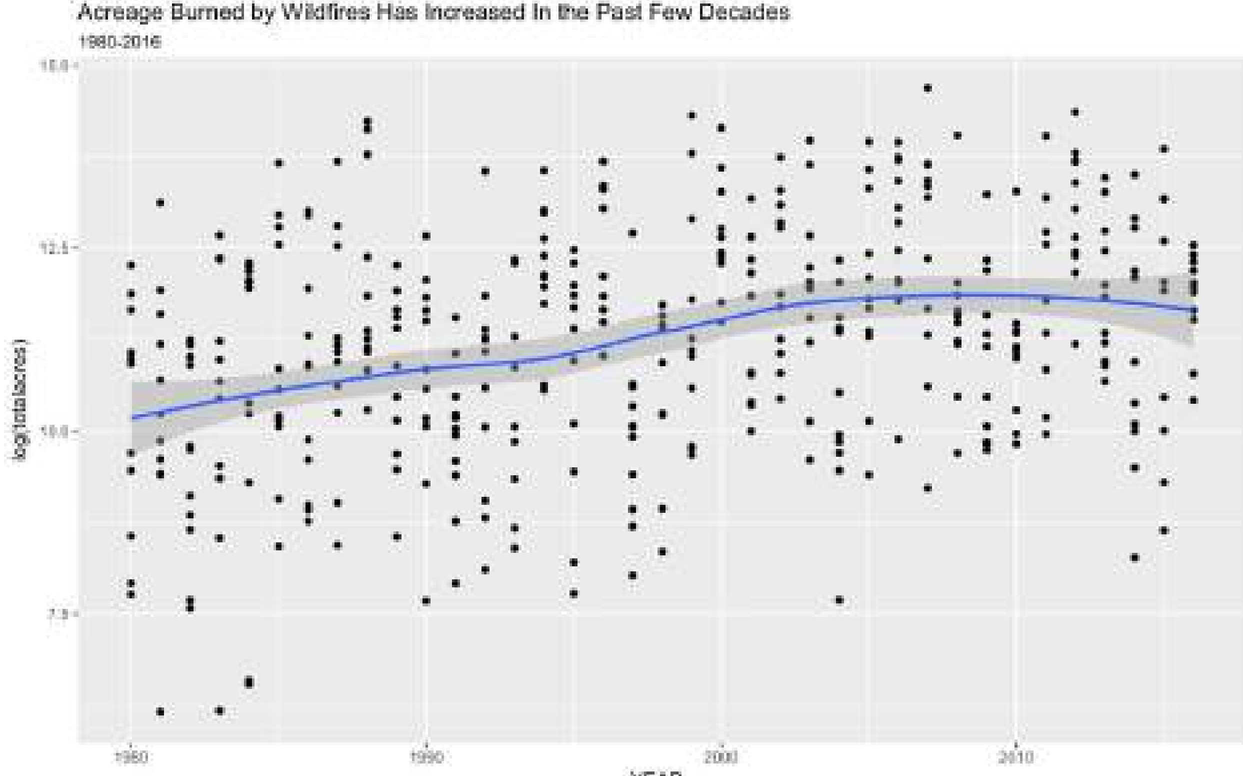
.

```
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() +
geom_smooth(method=loess, se=TRUE) + geom_text(aes(label=STATE,
color=STATE), size=2, check_overlap = TRUE, nudge_x = 1.0)
```



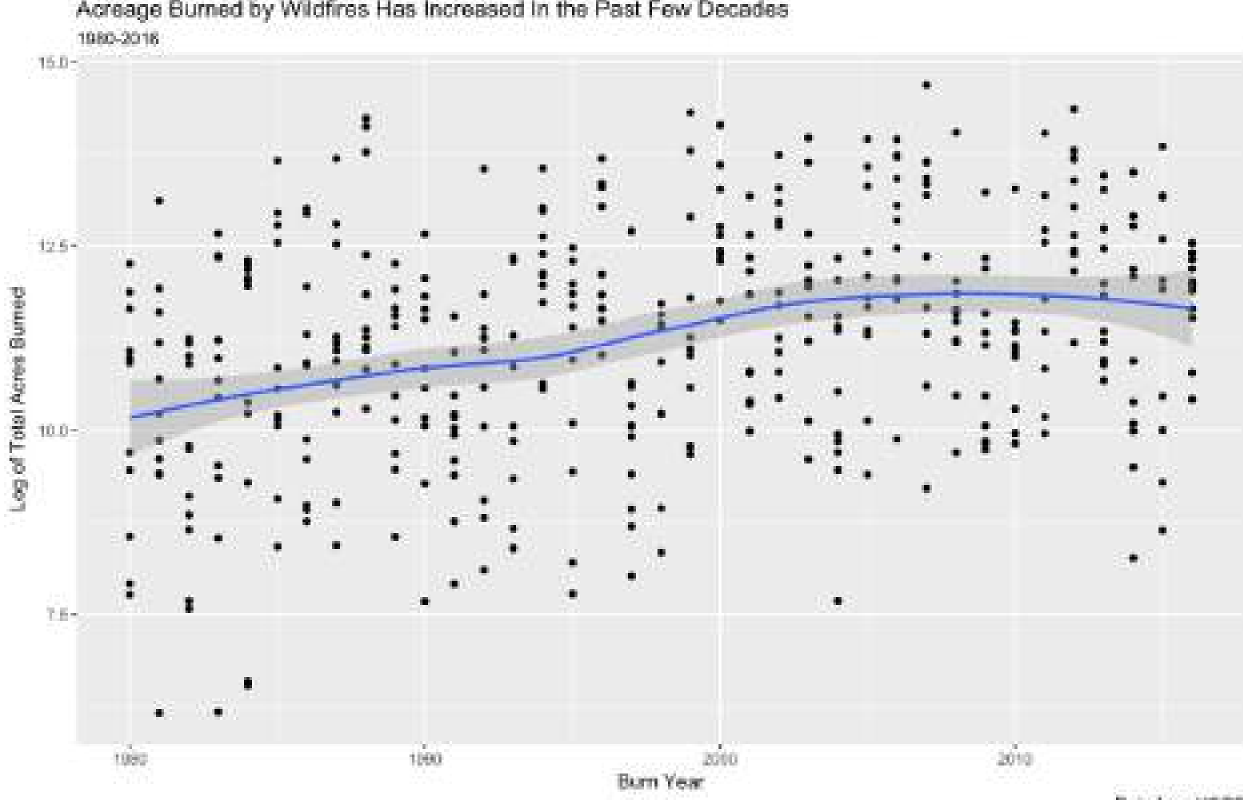
6. You can also add a subtitle and caption with the code you see below.

```
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() +
geom_smooth(method=loess, se=TRUE) + labs(title=paste(“Acreage Burned by
Wildfires Has Increased In the Past Few Decades”), subtitle=paste(“1980-
2016”), caption=”Data from USGS”)
```



7. You can also update the X and Y
labels for the graph. Update these labels on your graph using the code you see below.

```
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres))) + geom_point() +
geom_smooth(method=loess, se=TRUE) + labs(title=paste(“Acreage Burned by
Wildfires Has Increased In the Past Few Decades”), subtitle=paste(“1980-
2016”), caption=”Data from USGS”) + scale_y_continuous(name=”Log of Total
Acres Burned”) + scale_x_ continuous(name=”Burn Year”)
```



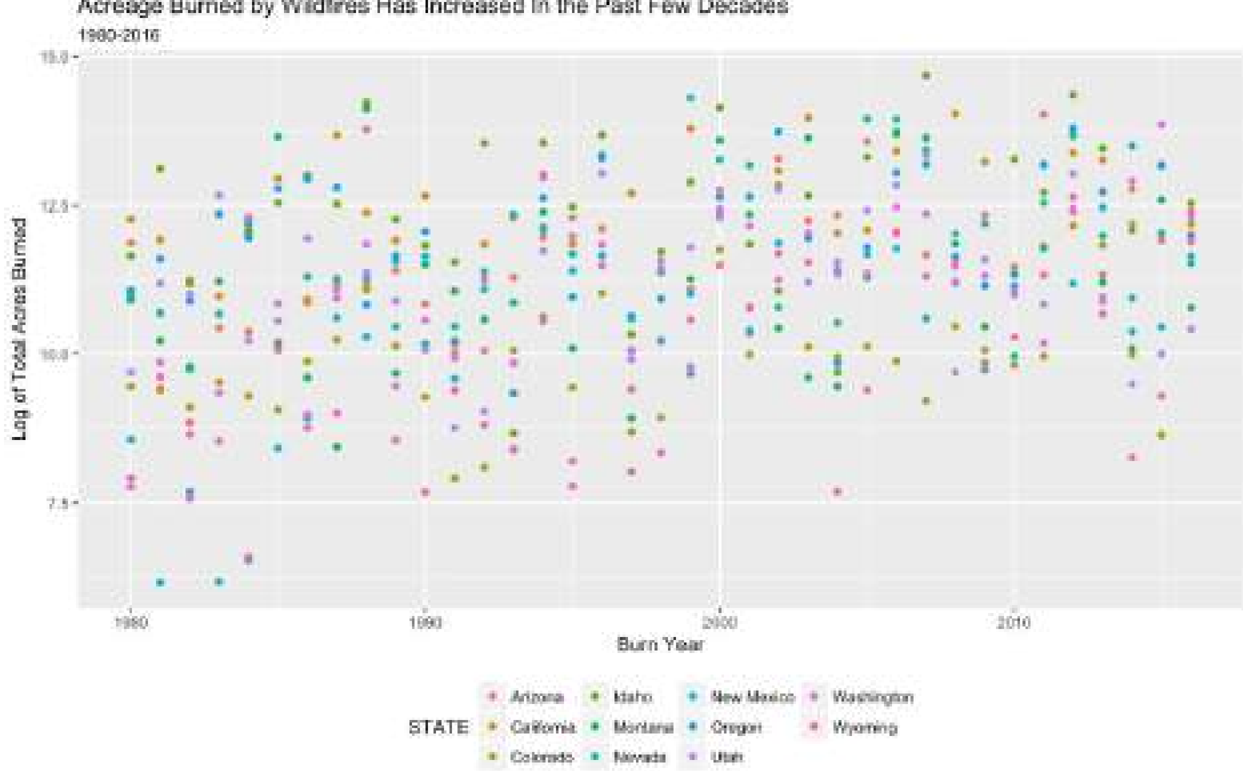
8. You can check your work against the solution file Chapter7_4.R

Step 5: Legend layouts

The theme() function can be used to control the location of the legend and the guides() function can be used to provide additional legend control.

1. The theme() function along with the legend.postion argument is used to control the location of the legend on the graph. By default, the legend we've seen so far has been placed on the right side of the graph with a vertical orientation. Reposition the legend to the bottom with the code below.

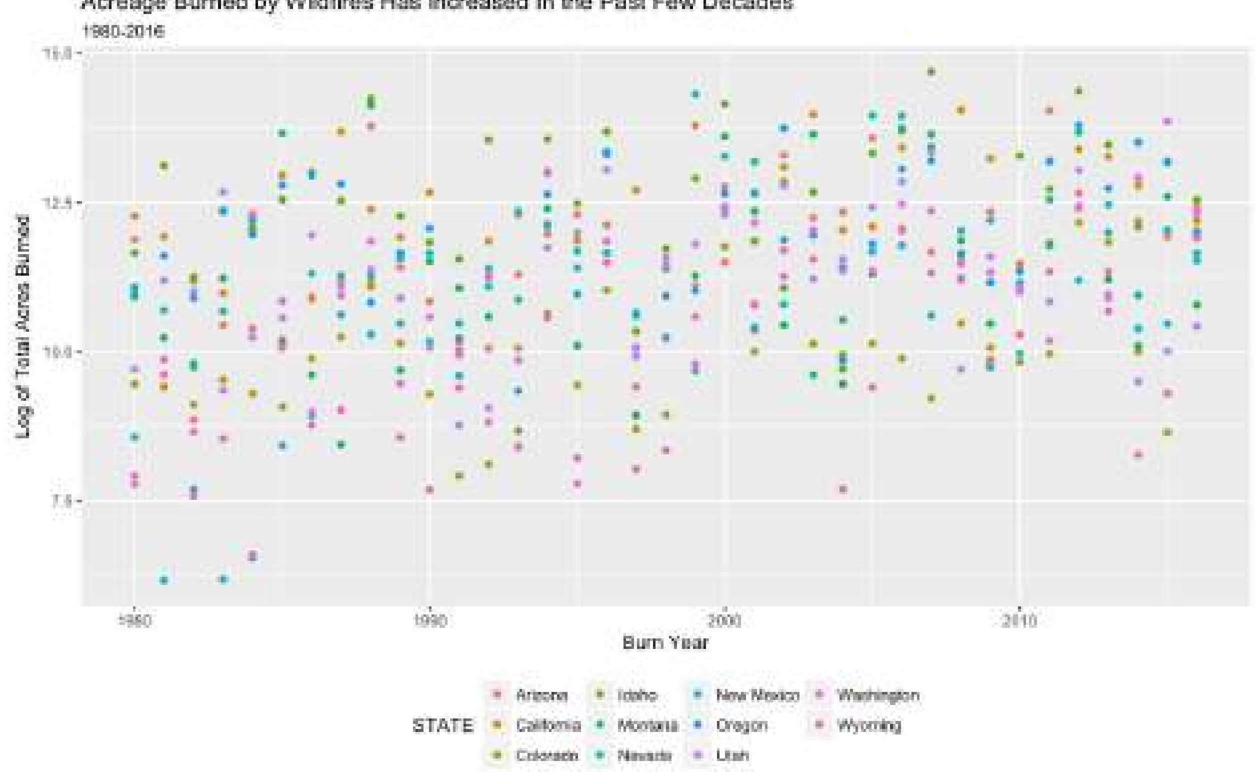
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres), color=STATE)) + geom_point() + labs(title=paste("Acreage Burned by Wildfires Has Increased In the Past Few Decades"), subtitle=paste("1980-2016"), caption="Data from USGS") + scale_y_continuous(name="Log of Total Acres Burned") + scale_x_continuous(name="Burn Year") + theme(legend.position="bottom")



2. You can also explicitly remove a legend by setting legend.position = “none”. Try that now if you’d like.

3. Other aspects of the legend such as the number of rows in the legend as well as the symbol size can be control through the guides() function. Use the code you see below to update the legend to be two rows and with each symbol set to size 4.

```
ggplot(data=sm, aes(x=YEAR_, y=log(totalacres), color=STATE)) + +
geom_point() +
+ labs(title=paste(“Acreage Burned by Wildfires Has Increased In the Past Few
Decades”), subtitle=paste(“1980-2016”), caption=”Data from USGS”) +
+ scale_y_continuous(name=”Log of Total Acres Burned”) + +
scale_x_continuous(name=”Burn Year”) +
+ theme(legend.position = “bottom”) +
+ guides(color=guide_legend(nrow=2,override.aes=list(size=4)))
```

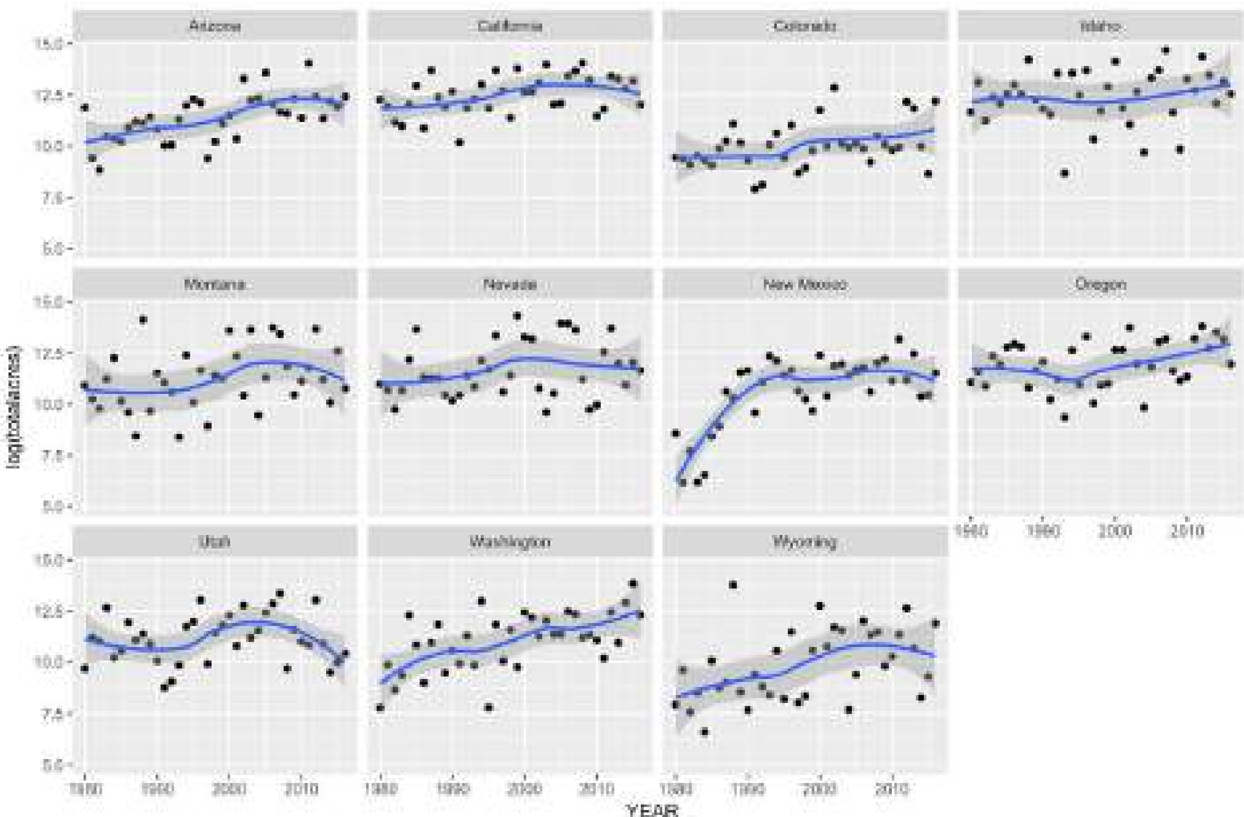


4. You can check your work against the solution file Chapter7_5.R

Step 6: Creating a facet

A particularly good way of graphing categorical variables is to split your plot into facets, which are subplots that each display one subset of the data. The facet_wrap() and facet_grid() function can be used to create facets.

1. Use the facet_wrap() function displayed in the code below to create a facet map that displays total acres burned by state.
ggplot(data=sm, mapping = aes(x=YEAR_, y=log(totalacres))) + geom_
point()+ facet_wrap(~STATE) + geom_smooth(method=loess, se=TRUE)



2. You can check your work against the solution file Chapter7_6.R

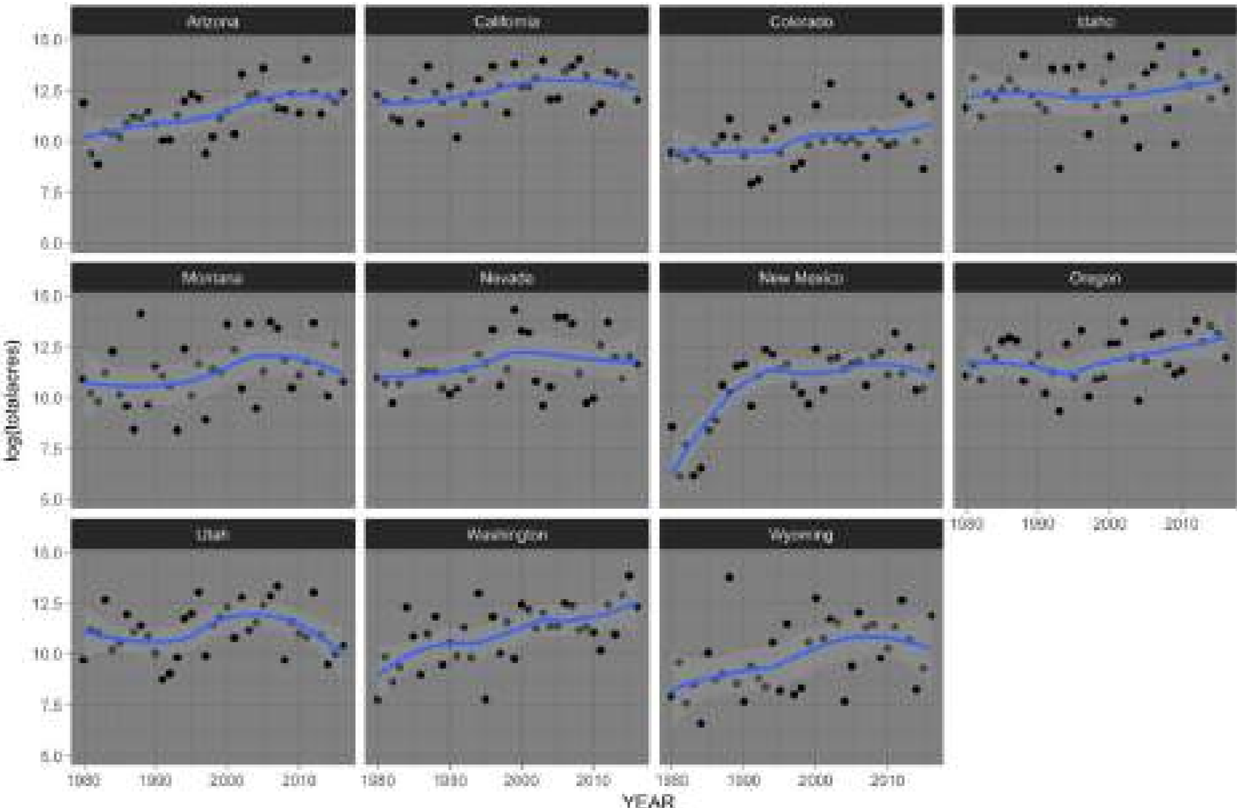
Step 7: Theming

includes eight built in themes that can be used to customize the styling of the
ggplot2
non-data elements of your plot.

1. The eight themes included in ggplot2 are theme_bw, theme_ classic, theme_dark, theme_gray, theme_light, theme_ linedraw, theme_minimal, theme_void.

Add the code you see below to change the facet to theme_dark
.

ggplot(data=sm, mapping = aes(x=YEAR_, y=log(totalacres))) + geom_point()+
facet_wrap(~STATE) + geom_smooth(method=loess, se=TRUE) + theme_dark()



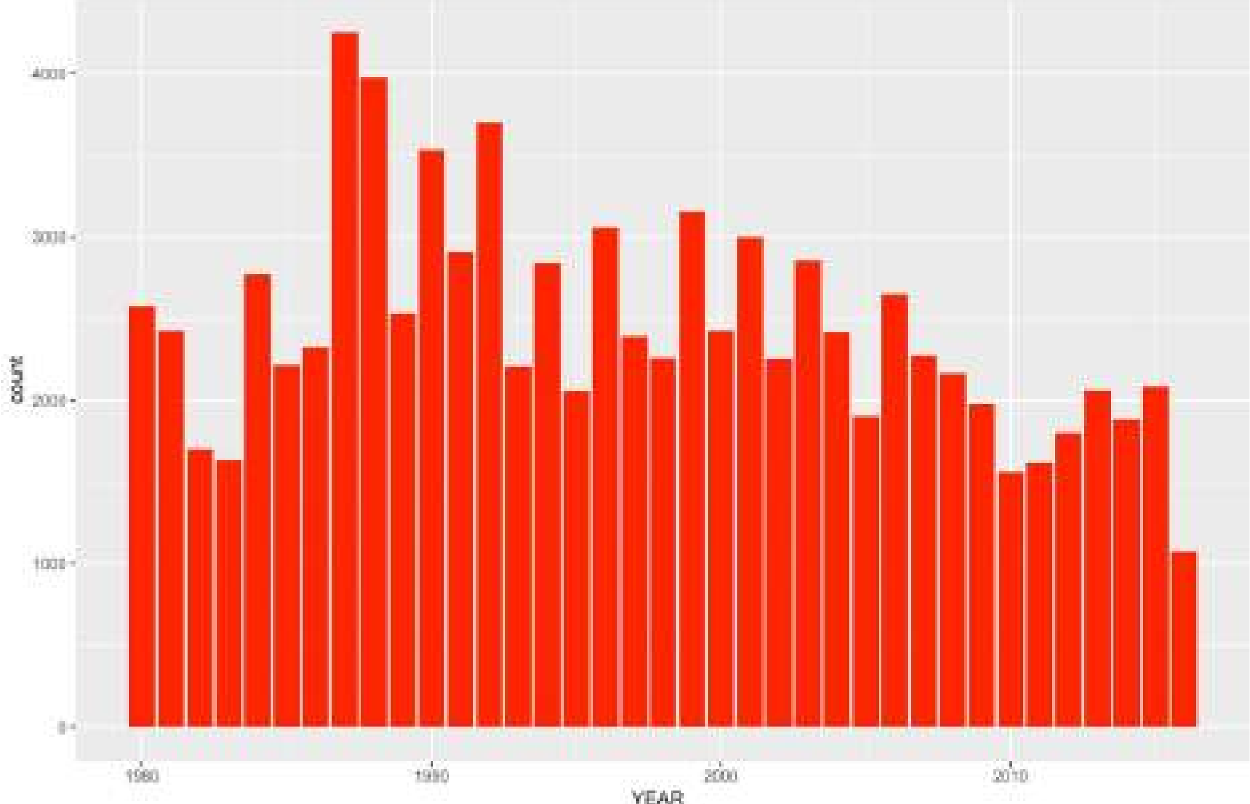
2. Experiment with the themes to see the differences in styling. 3. You can check your work against the solution file Chapter7_7.R

Step 8: Creating bar charts

You can use geom_bar() or geom_chart() to create bar charts with ggplot2. However, there is a significant difference between the two. The geom_bar() function will generate a count of the number of instances of a variable. In other words, it changes the statistic that has already been generated for the group. The geom_col() function keeps the variable already generated for the group. To see the difference, complete the following steps.

1. Load the StudyArea.csv file and get a subset of columns.
dfWildfires <- read_csv(“StudyArea.csv”, col_types = list(UNIT = col_character()), col_names = TRUE)
df <- select(dfWildfires, ORGANIZATI, STATE, YEAR_, TOTALACRES, CAUSE)
2. Filter the data frame so that only wildfires for California are included.
df <- filter(df, STATE == ‘California’)
3. Group the data frame by YEAR_.

grp <- group_by(df, YEAR_)
4. Plot the data using geom_bar() as seen below. Notice that the bar chart that is produced is a count of the number of fires for each year.
ggplot(data=grp) + geom_bar(mapping = aes(x=YEAR_), fill=”red”)



5. Now use geom_col() to see the difference. The TOTALACRES variable is maintained in this case.
ggplot(data=grp) + geom_col(mapping = aes(x=YEAR_, y=TOTALACRES), fill=”red”)
6. You can check your work against the solution file Chapter7_8.R

Step 9: Creating Violin Plots

Violin plots, which are similar to box plots, also show the probability density at various values. Thicker areas of the violin plot indicate a higher probability at that value. Typically, violin plots also include a marker for the median along with the Inter-Quartile Range (IQR). The geom_violin() function is used to create violin plots in ggplot2.

1. Load the StudyArea.csv file and get a subset of columns.
dfWildfires <- read_csv(“StudyArea.csv”, col_types = list(UNIT =

```
col_character()), col_names = TRUE)
df <- select(dfWildfires, ORGANIZATI, STATE, YEAR_, TOTALACRES,
CAUSE)
```

2. Filter the data frame so that only wildfires greater than 5,000 acres are included.

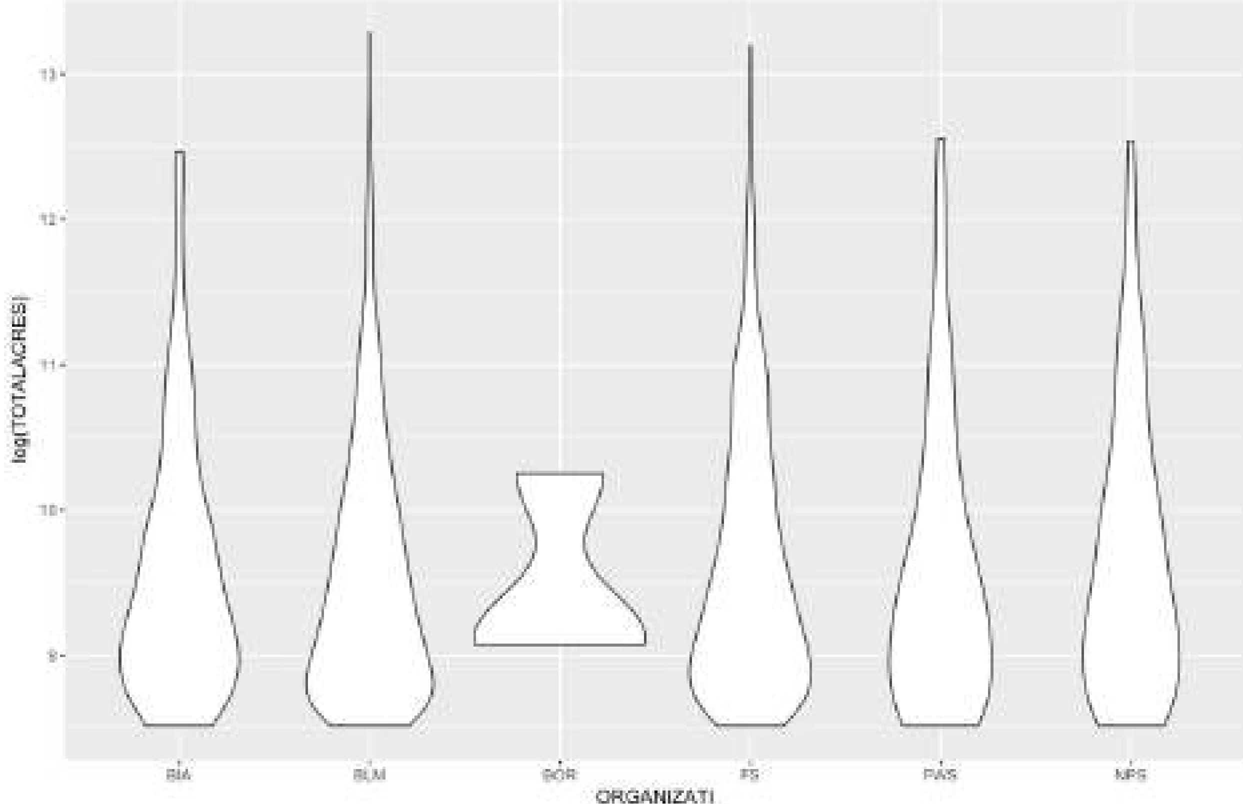
```
dfWildfires <- filter(dfWildfires, TOTALACRES >= 5000)
```

3. Group the wildfires by organization.

```
grpWildfires <- group_by(dfWildfires, ORGANIZATI)
```

4. Create a basic violin plot.

```
ggplot(data=grpWildfires, mapping = aes(x=ORGANIZATI,
y=log(TOTALACRES))) + geom_violin()
```



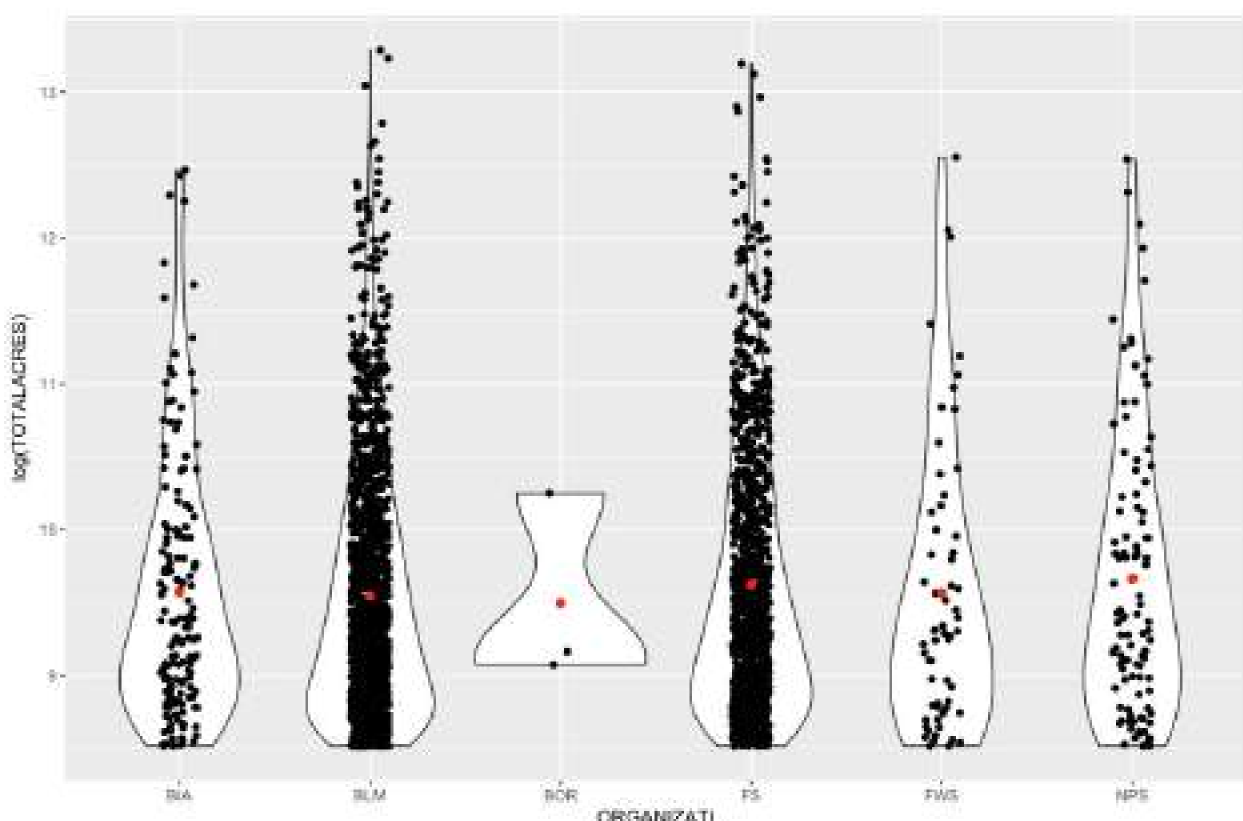
5. You can add the individual observations using geom_jitter().

```
ggplot(data=grpWildfires, mapping = aes(x=ORGANIZATI,
y=log(TOTALACRES))) + geom_violin() + geom_jitter(height = 0, width = 0.1)
```

6. The mean can be added using
stat_summary()
as seen below.

```
ggplot(data=grpWildfires, mapping = aes(x=ORGANIZATI,
```

y=log(TOTALACRES))) + geom_violin() + geom_jitter(height = 0, width = 0.1) + stat_summary(fun.y=mean, geom=”point”, size=2, color=”red”)



7. The box_plot() function can be used to add the mean and IQR. ggplot(data=grpWildfires, mapping = aes(x=ORGANIZATI, y=log(TOTALACRES))) + geom_violin() + geom_boxplot(width=0.1) 8. You can check your work against the solution file Chapter7_9.R

Step 10: Creating density plots

Density plots, created with geom_density() computes a density estimate, which is a smoothed version of a histogram and is used with continuous data. ggplot2 can also compute 2D versions of density includes contours and polygon styled density plots.

1. In this first portion of the exercise you’ll create a basic density plot. Load the StudyArea.csv file and get a subset of columns.
dfWildfires <- read_csv(“StudyArea.csv”, col_types = list(UNIT = col_character()), col_names = TRUE)
df <- select(dfWildfires, ORGANIZATI, STATE, YEAR_, TOTALACRES,

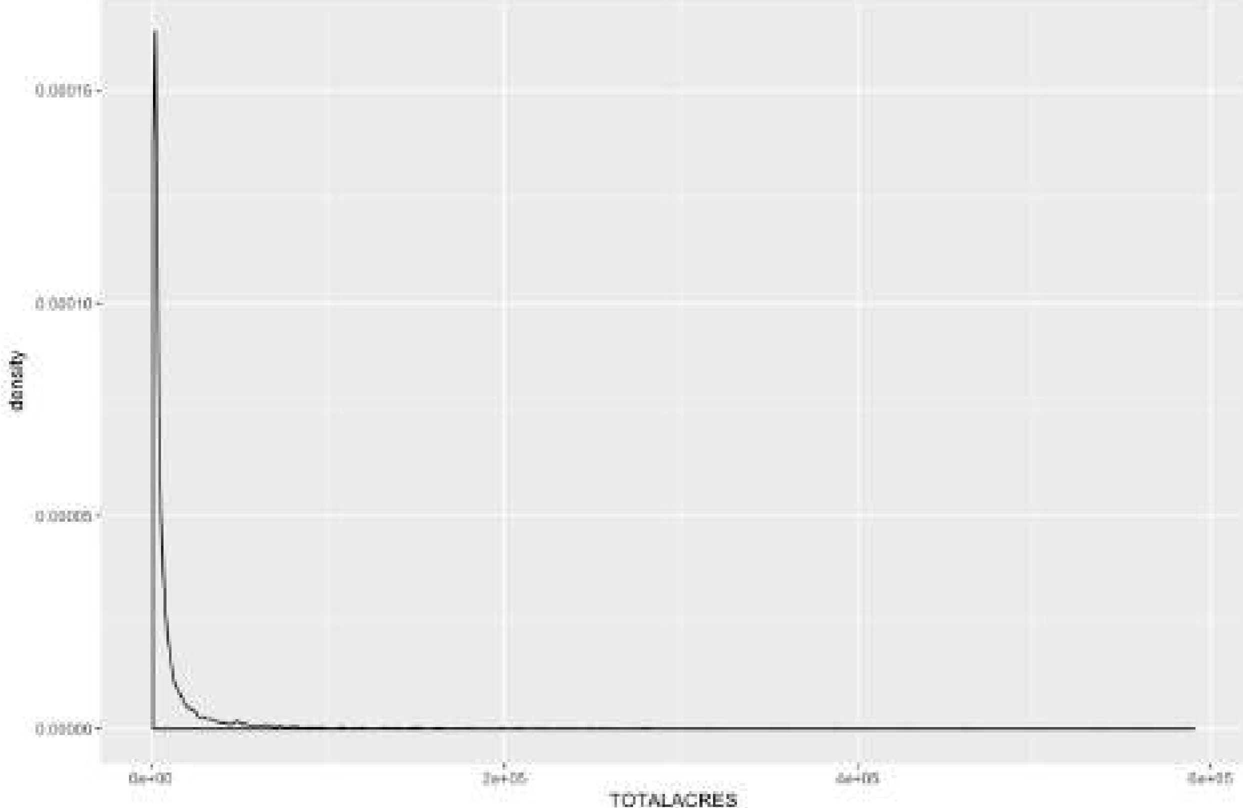
CAUSE)

2. Filter the data frame so that only wildfires greater than 1,000 acres are included.

```
dfWildfires <- filter(dfWildfires, TOTALACRES >= 1000)
```

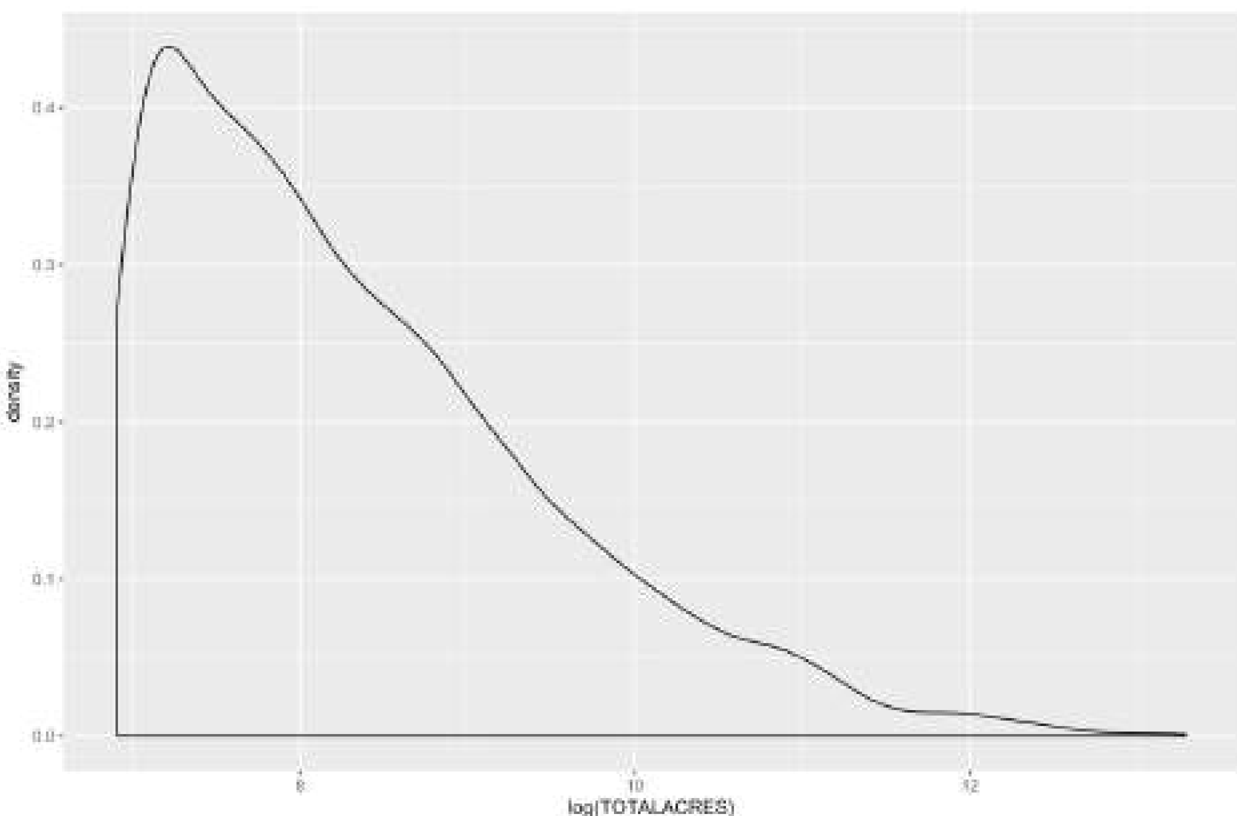
3. Create a density plot with the geom_density() function.

```
ggplot(dfWildfires, aes(TOTALACRES)) + geom_density()
```



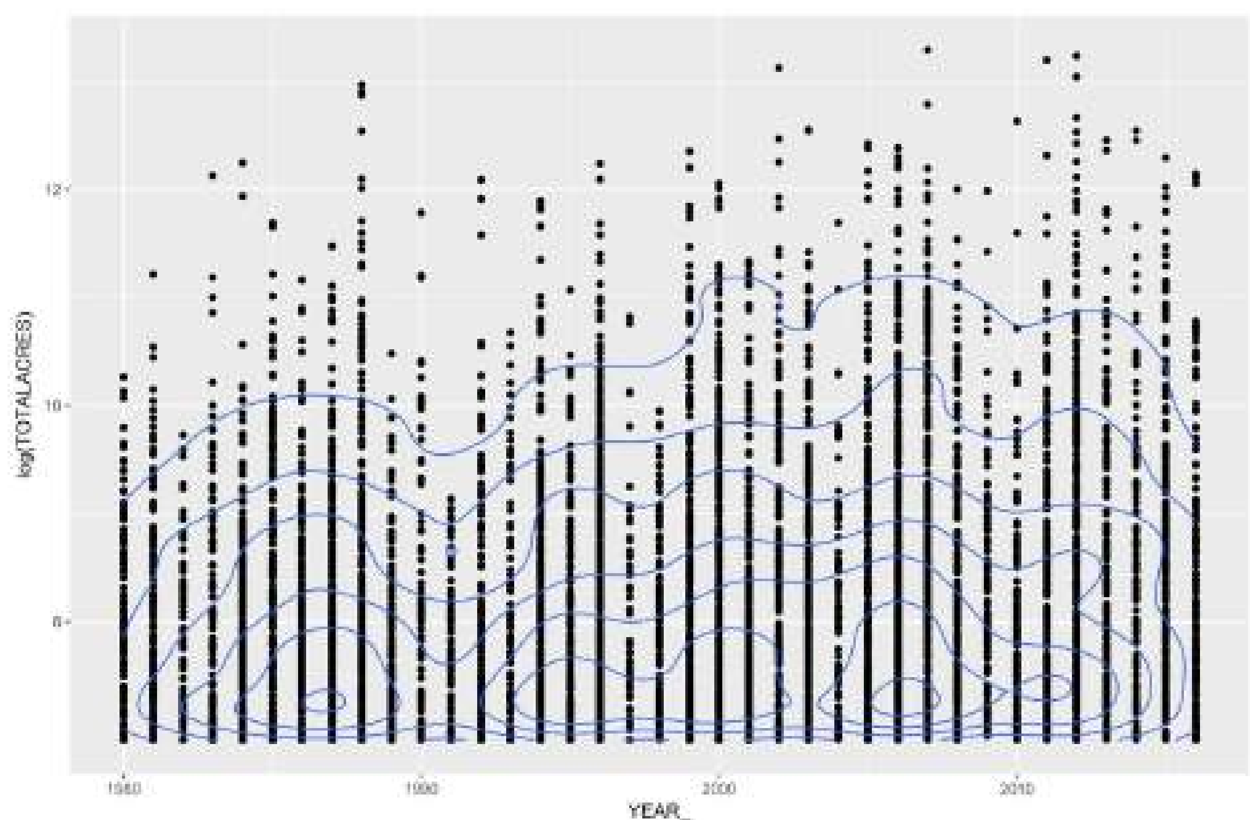
4. You may also want to create the same density plot with a logged version of the data.

```
ggplot(dfWildfires, aes(log(TOTALACRES))) + geom_density()
```



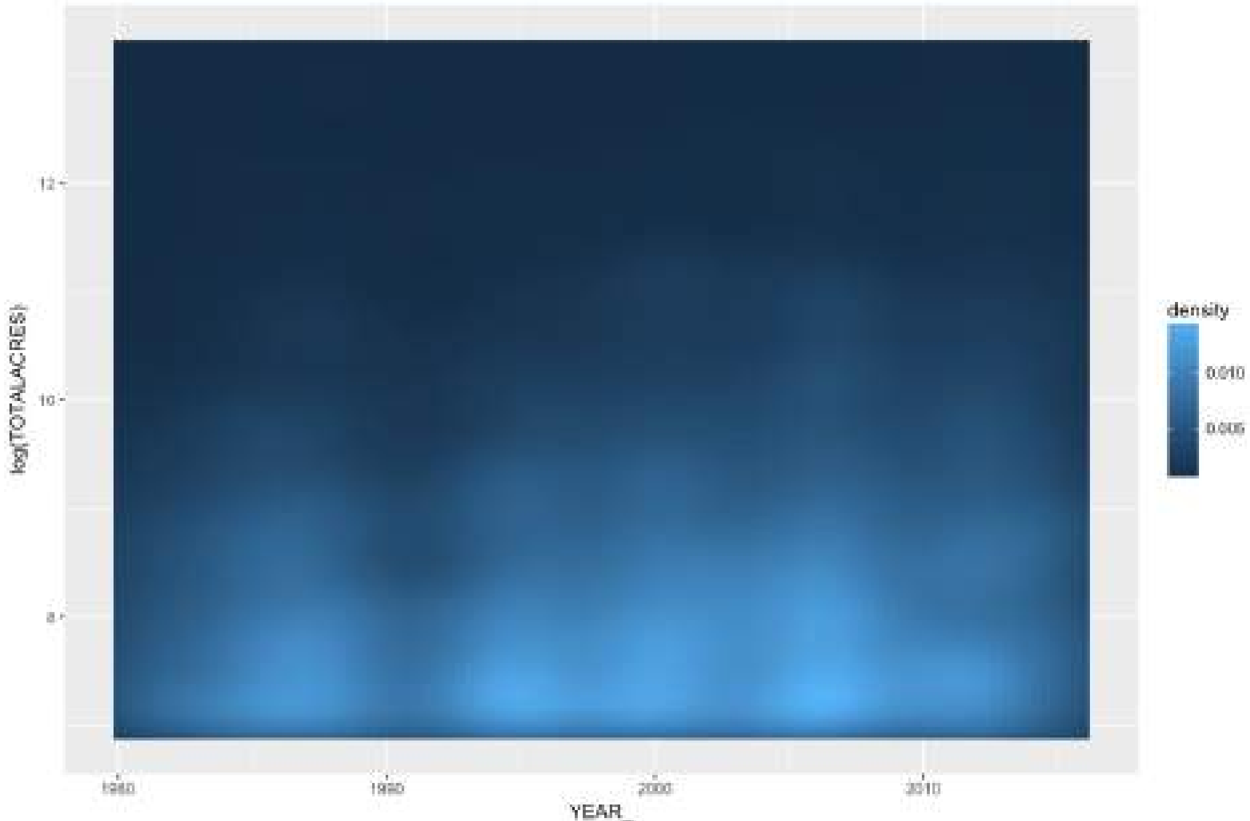
5. Next, you’ll create 2D plots of the data starting with contours. Add the code you see below.

ggplot(dfWildfires, aes(x=YEAR_, y=log(TOTALACRES))) + geom_point() + geom_density_2d()



6. Finally, create a 2D density surface using stat_density_2d().

```
ggplot(dfWildfires, aes(x=YEAR_, y=log(TOTALACRES))) + geom_
density_2d() + stat_density_2d(geom=”raster”, aes(fill=.. density..),
contour=FALSE)
```



7. You can check your work against the solution file Chapter7_10.R

Conclusion

In this chapter you learned various data visualization techniques using ggplot2. We started with basic scatterplots, added regression lines, labeled the graphs in various ways, and created a legend. In addition, you learned how to create facet plots, and work with ggplot2s built in theming options. You also learned how to create bar charts, violin charts, and density plots.

In the next chapter you will learn how to create maps using the ggmap package.

Chapter 8

Visualizing Geographic Data with ggmap

The ggmappackage enables the visualization of spatial data and spatial statistics in a map format using the layered approach of ggplot2. This package also includes basemaps that give your visualizations context including Google Maps, Open Street Map, Stamen Maps, and CloudMade maps. In addition, utility functions are provided for accessing various Google services including Geocoding, Distance Matrix, and Directions.

The ggmap package is based on ggplot2, which means it will take a layered approach and will consist of the same five components found in ggplot2. These include a default dataset with aesthetic mappings where x is longitude, y is latitude, and the coordinate system is fixed to Mercator. Other components include one or more layers defined with a geometric object and statistical transformation, a scale for each aesthetic mapping, coordinate system, and facet specification. Because ggmap is built on ggplot2 is has access to the full range of ggplot2 that you learned about in a previous exercise.

In this chapter we'll cover the following topics:

- Creating a basemap
- Adding operational layers
- Adding layers from a shapefile

Exercise 1: Creating a basemap

There are two basic steps to create a map with ggmap. The details are more complex than these two steps might imply, but in general you just need to download the map raster (basemap) and then plot operational data on the basemap. The first step is to download the map raster, also known as the basemap. This is accomplished using the get_map() function, which can be used to create a basemap from Google, Stamen, Open Street Map, or CloudMade. You'll learn how to do that in this step. In a future step you'll learn how to add and style operational data in various ways.

1. Open RStudio and find the **Console** pane.

2. If necessary, set the working directory by typing the code you see below into

the **Console** pane or by going to **Session | Set Working Directory | Choose Directory** from the RStudio menu.

```
setwd(<installation directory for exercise data>)
```

3. Load the ggmap package by going to the **Packages** pane in RStudio and clicking on the checkbox next to the package name. Alternatively, you can load it from the **Console** by typing:

```
library(ggmap)
```

4. Create a variable called myLocation and set it to California.

```
myLocation <- "California"
```

5. Call the get_map() function and pass in the location variable along with a zoom level of 6.

```
myMap <- get_map(location = myLocation, zoom = 6)
```

6. In RStudio you should see some return messages that look similar to the code you see below. If you don't see something similar to this, you may need to re-execute the script. It isn't uncommon to get an error message when calling the get_map() function from R Studio. If this happens simply re-execute the code until you get something that is similar to what you see below.

```
Map from URL: http://maps.googleapis.com/maps/api/staticmap?center
=California&zoom=6&size=640x640&scale=2&maptype=terrain&language=
en-EN&sensor=false
Information from URL : http://maps.googleapis.com/maps/api/ geocode/json?
address=California&sensor=false
```

7. Call the ggmap() function, passing in the myMap variable. The **Plots** pane should display the map as seen below. The default map type is Google Maps with a style of Terrain.

```
ggmap(myMap)
```

The Google source includes a number of map types including those you see in the screenshot below.

8. Add and execute the code you see below to add a Google satellite map.

```
myMap <- get_map(location = myLocation, zoom = 6, source=”google”,
maptype=”satellite”)
ggmap(myMap)
```

9. There are a number of ways that you can define the input location: longitude/latitude coordinate pair, a character string, or a bounding box. The character string tends to be a more practical solution in many situations since you can simply pass in the name of the location. For example, you could define

the location as Houston Texas or The White House or The Grand Canyon. When a character string is passed to the location parameter it is then passed to the geocoding service to obtain the latitude/longitude coordinate pair. Add the code you see below to see how passing in a character string works.

```
myMap <- get_map(location = “Grand Canyon, Arizona”, zoom = 11)
ggmap(myMap)
```

The zoom level can be set between 3 and 21 with 3 representing a continent level view, and 21 representing a building level view. Take some time to

experiment with the zoom level to see the effect of various settings.

10. You can check your work against the solution file Chapter8_1.R

Exercise 2: Adding operational data layers

ggmap() returns a ggplot object, meaning that it acts as a base layer in the ggplot2 framework. This allows for the full range of ggplot2 capabilities meaning that you can plot points on the map, add contours and 2D heat maps, and more. We'll examine some of these capabilities in this section.

1. Initially we'll just load the wildfire events as points. Add the code you see below to produce a map of California that displays wildfires from the years 1980-2016 that burned more than 1,000 acres.

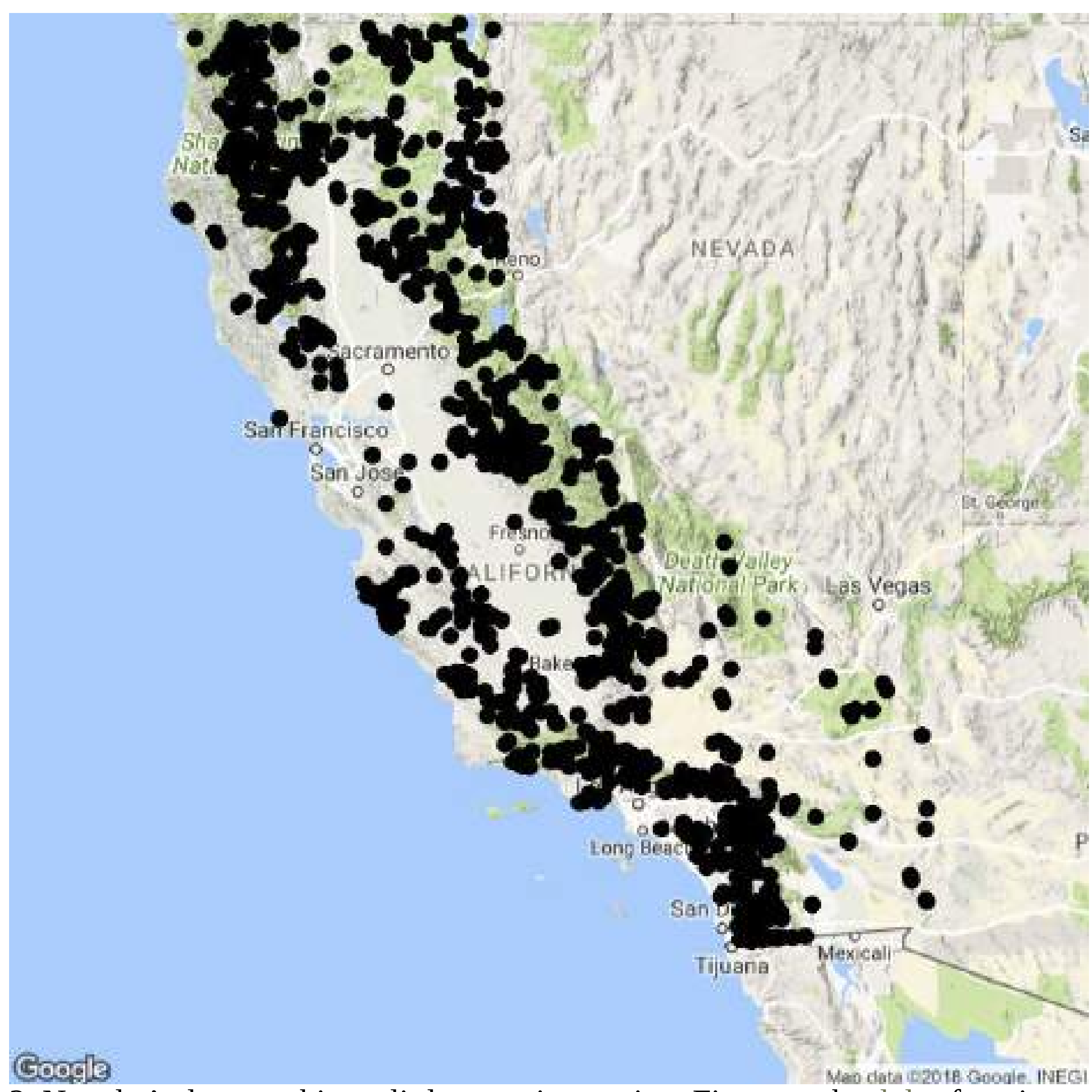
```
myLocation <- “California”
#get the basemap layer
myMap <- get_map(location = myLocation, zoom = 6)

#read in the wildfire data to a data frame (tibble) dfWildfires <-
read_csv(“StudyArea_SmallFile.csv”, col_names = TRUE)

#select specific columns of information
df <- select(dfWildfires, STATE, YEAR_, TOTALACRES, DLATITUDE,
DLONGITUDE)

#filter the data frame so that only fires greater than 1,000 acres burned in
California are present
df <- filter(df, TOTALACRES >= 1000 & STATE == ‘California’)

#use geom_point() to display the points. The x and y properties of the aes()
function are used to define the geometry
ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y =
DLATITUDE))
```



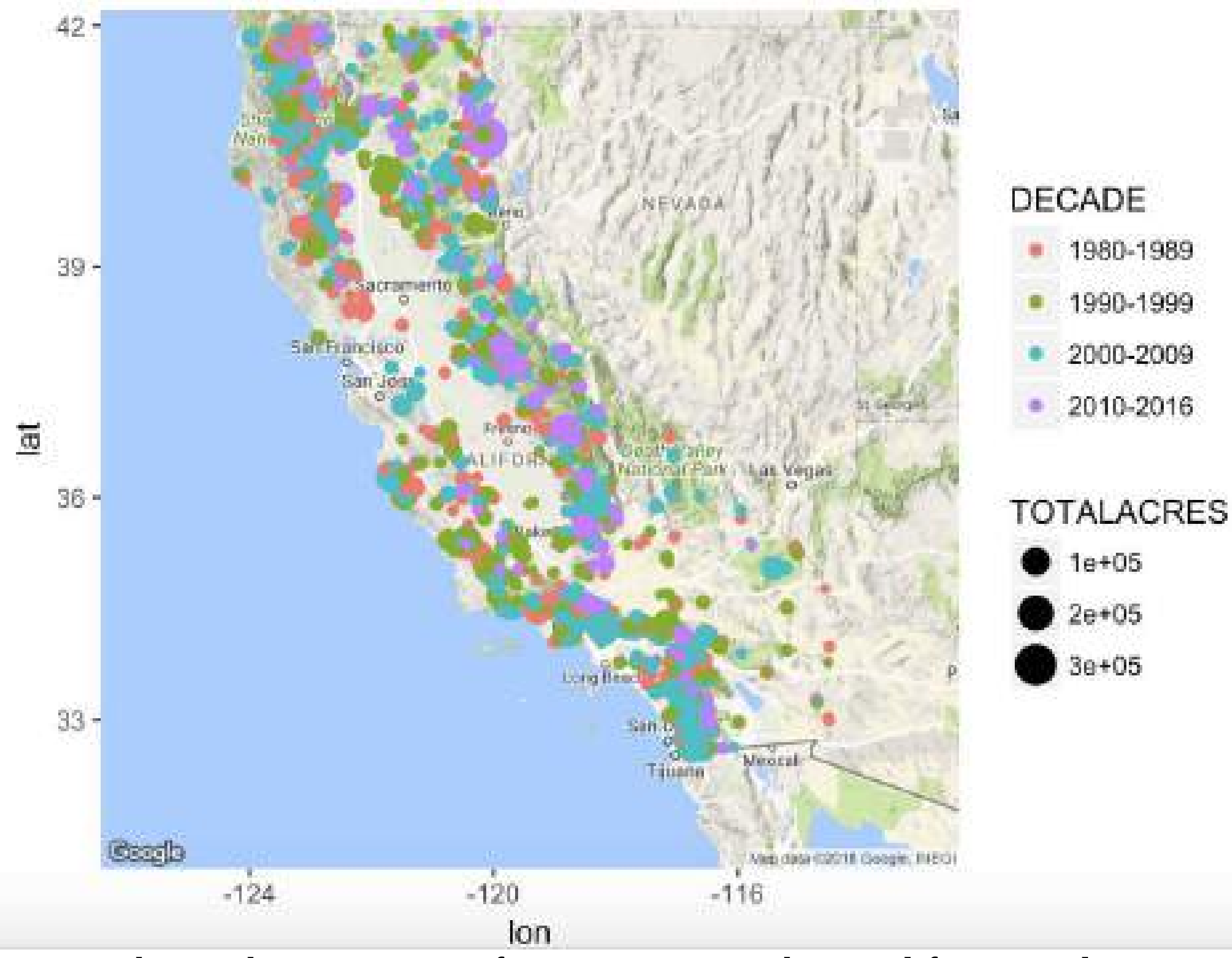
2. Now let's do something a little more interesting. First, use the dplyr function mutate()
to group the fires by decade.

to group the fires by decade.
1989", ifelse(YEAR_ %in% 1990:1999, "1990-1999", ifelse(YEAR_ %in% 2000:2009, "2000-2009", ifelse(YEAR_ %in% 2010:2016, "2010-2016", "-99")))))

3. Next, color code the wildfires by DECADE and create a graduated symbol map based on the size of each fire. The colour property defines the column to

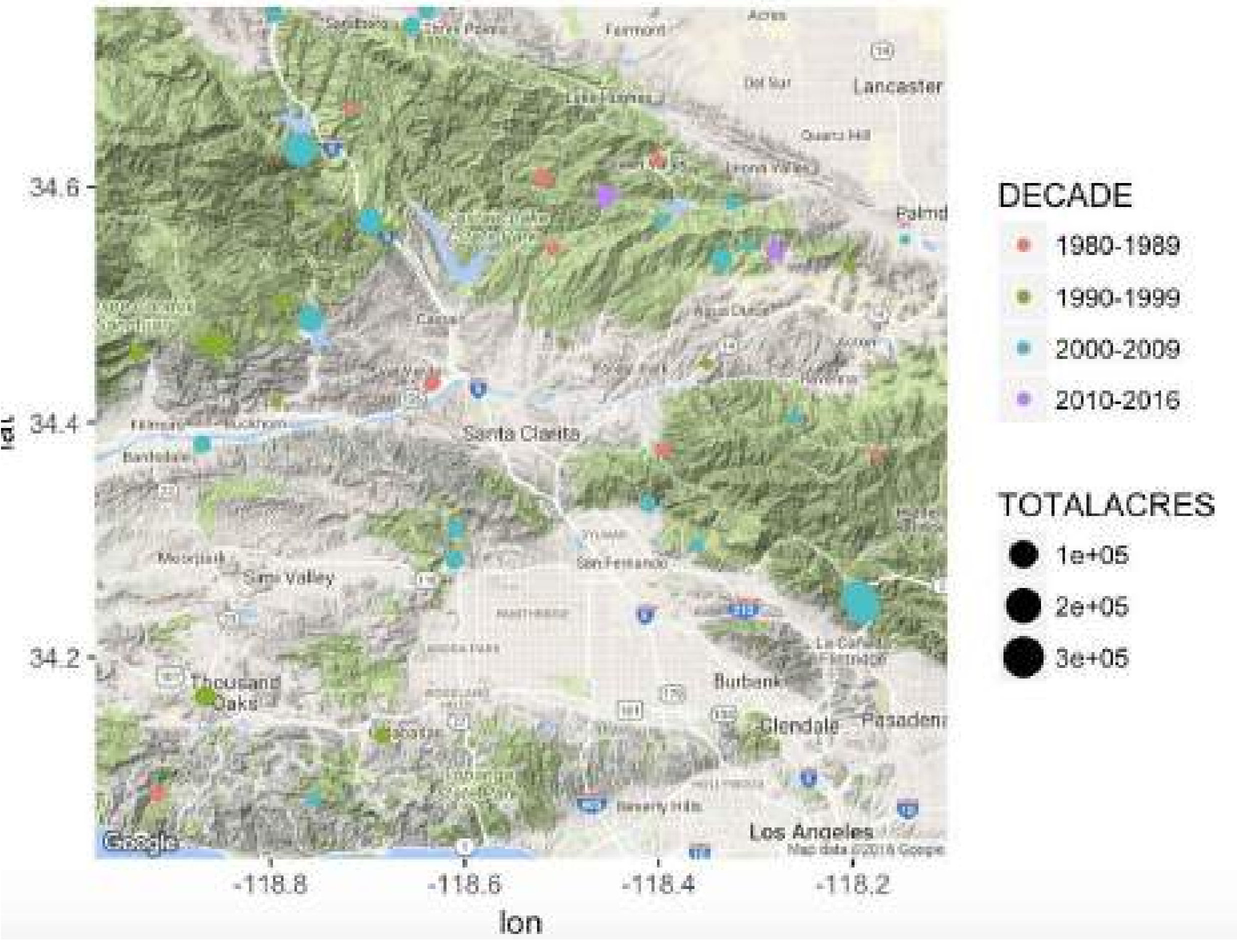
use for grouping, and the size property define the column to use for the size of each symbol.

ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y = DLATITUDE, colour= DECADE, size = TOTALACRES))
This should produce a map that appears as seen in the screenshot below.



4. Let's change the map view to focus more on southern California, and in particular the area just north of Los Angeles.

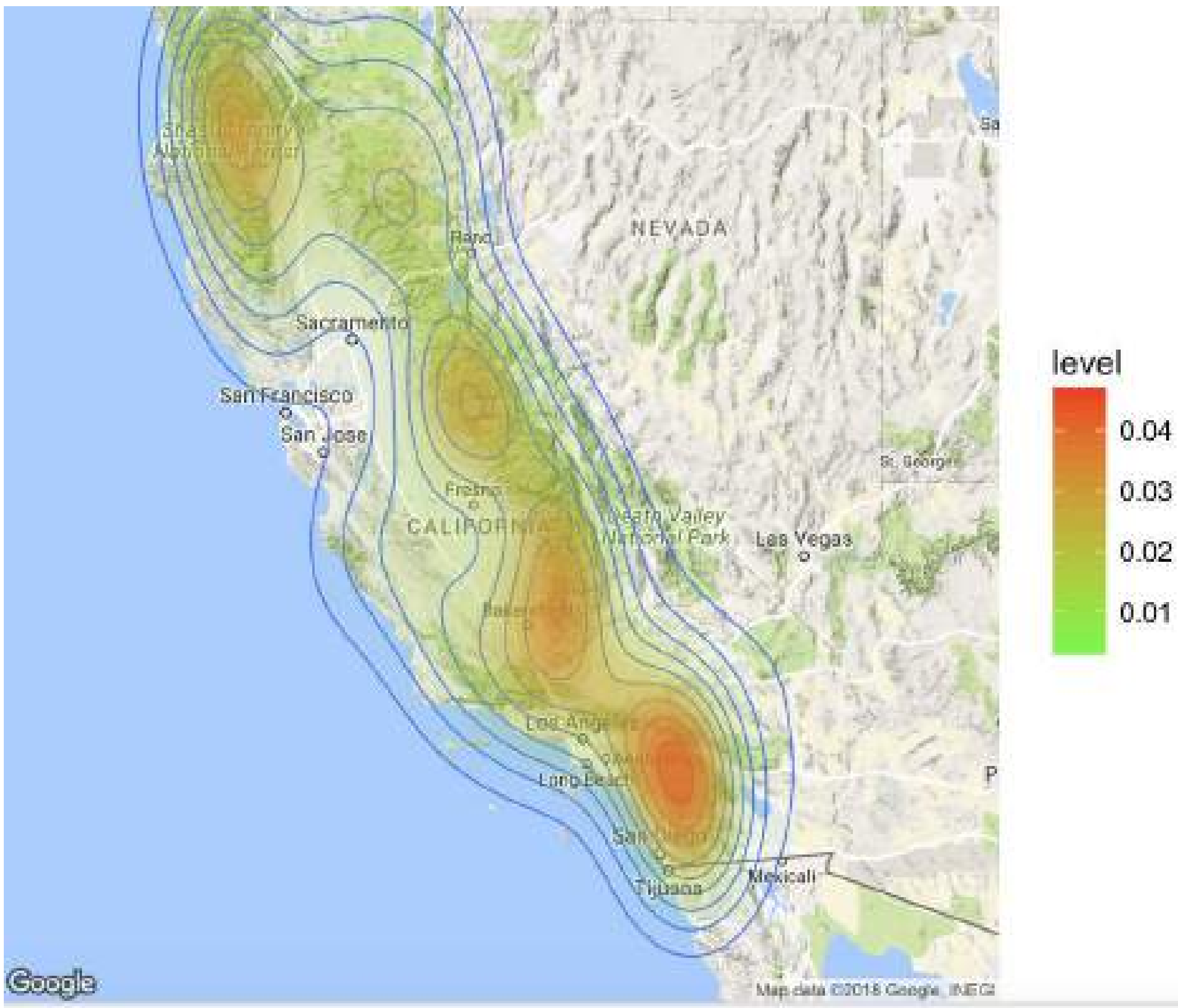
myMap <- get_map(location = "Santa Clarita, California", zoom = 10)
ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y = DLATITUDE, colour= DECADE, size = TOTALACRES))



5. Next, we'll add contour and heat layers. The g eom_density2d() function is used to create the contours while the stat_density2d() function creates the heat map. Add the following code to produce the map you see below. You can experiment with the colors using the scale_fill_ gradient(low and high) properties. Here we've set them to green and red respectively, but you may want to change the color scheme.

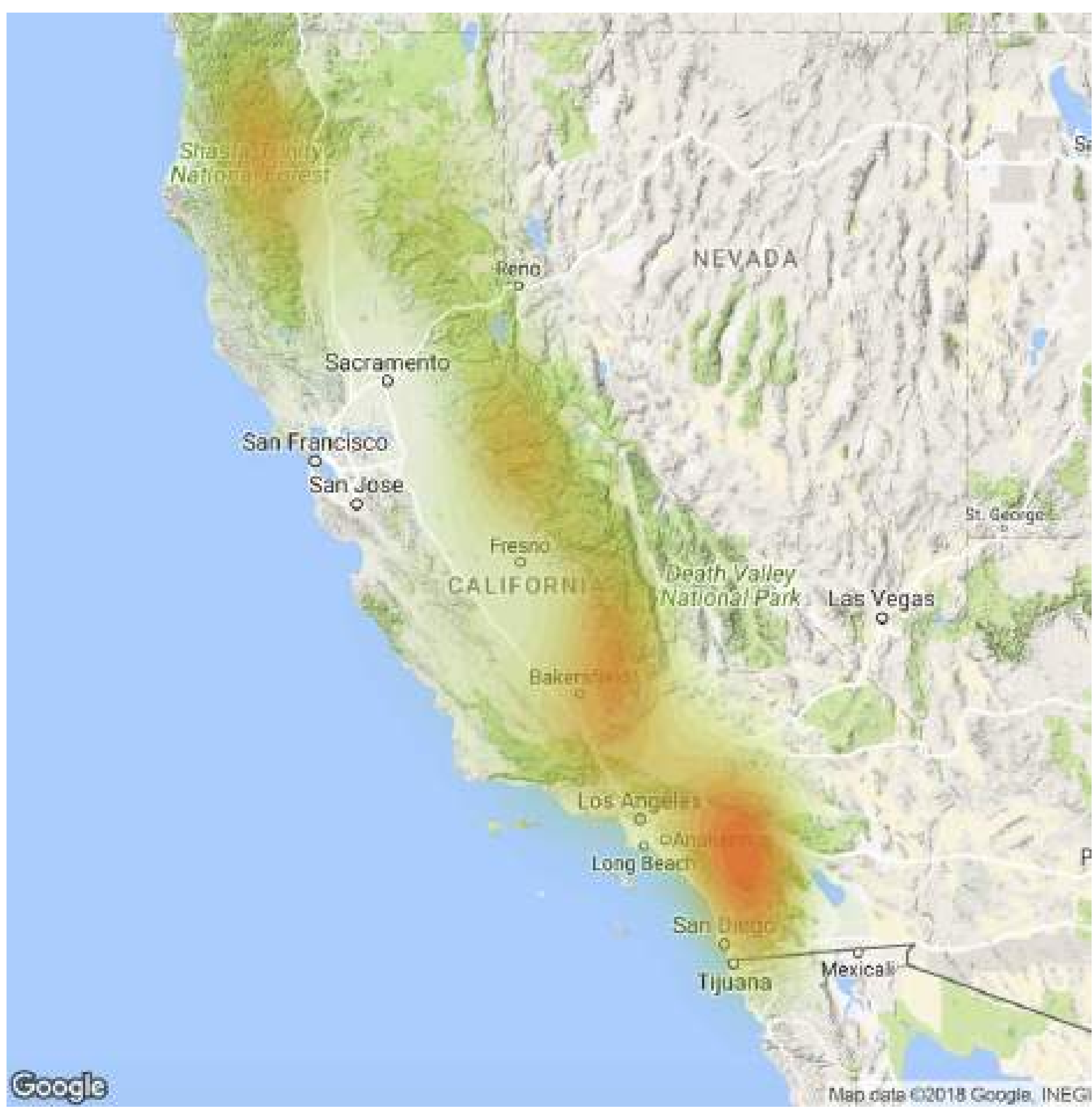
myMap <- get_map(location = "California", zoom = 6)

ggmap(myMap, extent = "device") + geom_density2d(data = df, aes(x = DLONGITUDE, y = DLATITUDE), size = 0.3) + stat_density2d(data = df, aes(x = DLONGITUDE, y = DLATITUDE, fill = ..level.., alpha = ..level..), size = 0.01, bins = 16, geom = "polygon") + scale_fill_ gradient(low = "green", high = "red") + scale_alpha(range = c(0, 0.3), guide = FALSE)



6. If you’d prefer to see the heat map without contours, the code can be simplified as follows:

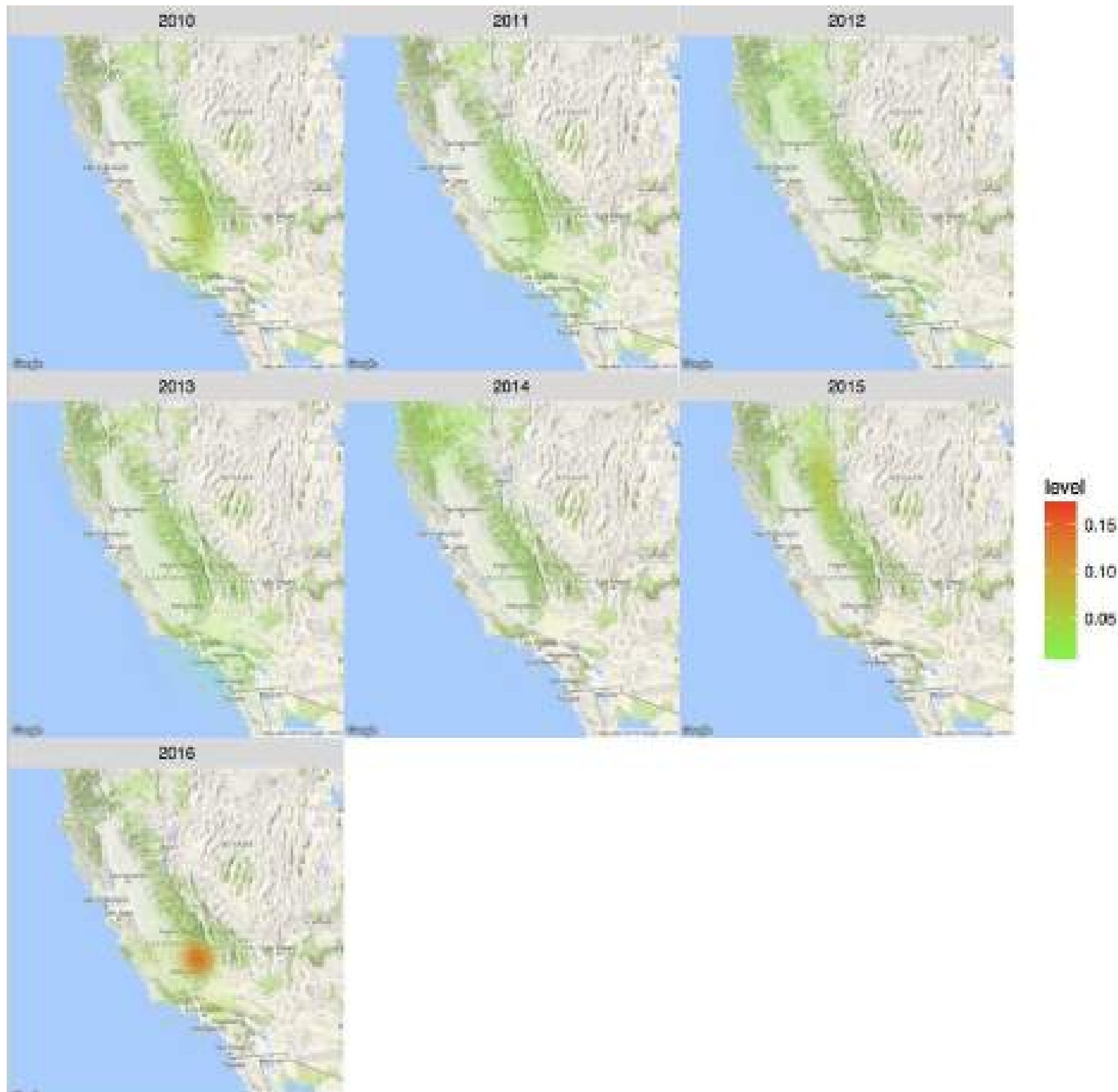
ggmap(myMap, extent = “device”) + stat_density2d(data = df, aes(x = DLONGITUDE, y = DLATITUDE, fill = ..level.., alpha = ..level..), size = 0.01,bins = 16, geom = “polygon”) + scale_fill_gradient(low = “green”, high = “red”) + scale_alpha(range = c(0, 0.3), guide = FALSE)



7. Finally, let's create a facet map that depicts hot spots for each year in the current decade. Add the following code to see how this works. The dataset contains information up through the year 2016.

```
df <- filter(df, YEAR_ %in% c(2010, 2011, 2012, 2013, 2014, 2015, 2016))

ggmap(myMap, extent = “device”) + stat_density2d(data = df, aes(x =
DLONGITUDE, y = DLATITUDE, fill = ..level.., alpha = ..level..), size =
0.01,bins = 16, geom = “polygon”) + scale_fill_gradient(low = “green”, high =
“red”) + scale_alpha(range = c(0, 0.3), guide = FALSE) + facet_wrap(~ YEAR_)
```



8. You can check your work against the solution file Chapter8_2.R

Exercise 3: Adding Layers from Shapefiles

While they are somewhat of an older GIS data format, shapefiles are still commonly used to represent geographic features. With a little bit of manipulation, you can get plot data from shapefiles onto ggmap.

1. For this exercise you'll need to install an additional package called rgdal. Use the **Packages** pane to find and install rgdal or enter the code you see below.

```
install.packages("rgdal")
```

2. Load the rgdal package through the **Packages** pane or enter the code you see below.

```
library(rgdal)
```

3. The Data folder that contains the exercise data for this book contains a shapefile called S_USA.Wilderness. You'll actually see a number of files with this name, but a different file extension. These files combine to create what is called a shapefile. This file contains the boundaries of designated wilderness areas in the United States. Use the readOGR() function from rgdal to load the data into a variable.

```
wild = readOGR('.', 'S_USA.Wilderness')
```

4. The fortify() function, which is part of ggplot2, converts all the individual points that define each boundary into a data frame that can then be used to plot the polygon boundaries.

```
wild <- fortify(wild)
```

5. Use the ggmap qmap() function (qmap means quick map) to create the basemap that will be used as the reference for the wilderness boundaries. Center the map in Montana.

```
montana <- qmap("Montana", zoom=6)
```

6. Before plotting the wilderness boundaries as polygons on the map, take a look at the data frame that was created by the fortify() function so you'll have a better understanding of the structure created by this function.

```
View(wild)
```

	long	lat	order	hole	piece	id	group
1	-70.94738	44.27397	1	FALSE	1	0	0.1
2	-70.94757	44.27229	2	FALSE	1	0	0.1
3	-70.94924	44.27145	3	FALSE	1	0	0.1
4	-70.95507	44.27034	4	FALSE	1	0	0.1
5	-70.95729	44.26868	5	FALSE	1	0	0.1
6	-70.95862	44.26679	6	FALSE	1	0	0.1
7	-70.95924	44.26590	7	FALSE	1	0	0.1
8	-70.96341	44.26423	8	FALSE	1	0	0.1
9	-70.96951	44.26340	9	FALSE	1	0	0.1
10	-70.97023	44.26174	10	FALSE	1	0	0.1
11	-70.97035	44.26145	11	FALSE	1	0	0.1
12	-70.97313	44.26062	12	FALSE	1	0	0.1
13	-70.97627	44.26080	13	FALSE	1	0	0.1
14	-70.97813	44.26090	14	FALSE	1	0	0.1
15	-70.97881	44.26107	15	FALSE	1	0	0.1

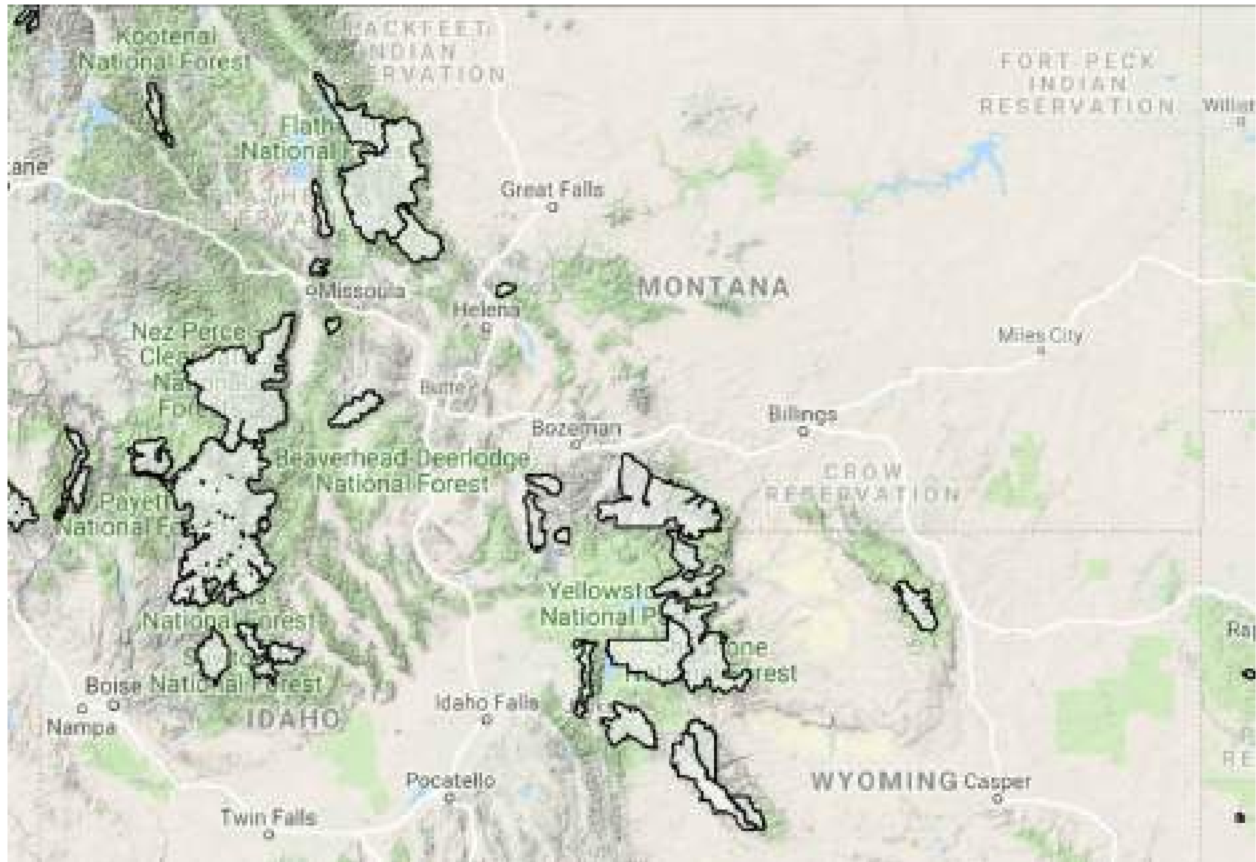
Take a look at the group column. This column uniquely identifies each wilderness boundary. The wilderness boundaries are polygons, and polygons are defined by a set of points which define the structure of the polygon. It’s sort of like playing connect the dots, where each dot is a latitude/longitude coordinate pair defined by the long and lat columns in the data frame.

For example, take a look at group 0.1. Notice that there are multiple rows that contains the value 0.1, and that each row has unique long and lat values. These are all the points used to define the boundaries of that polygon.

7. Now plot the wilderness boundaries on the basemap. Notice the use of the group column for grouping the polygons. It does take some time to plot the boundaries on the map so be patient with this step. Eventually you should see a map similar to the screenshot below.

montana + geom_polygon(aes(x=long,y=lat, group=group, alpha=0.25), data=wild, fill=’white’) + geom_polygon(aes(x=long,y=lat, group=group),

data=wild, color='black', fill=NA)



8. Optional – Use the color, fill, and alpha (used to define transparency) parameters to change the symbology to different colors and styles. 9. You can check your work against the solution file Chapter8_3.R

Conclusion

In this chapter you learned how to use the ggmap package to create compelling data visualizations in map format. You learned how to created basemaps using Google as a data source, add operational data layers, create various types of map visualizations using external data sources, and load shapefiles.

In the next chapter you will learn how to use R Markdown to share your work with others.

Chapter 9

R Markdown

R Markdown is an authoring framework for data science that combines code, results, and commentary. Output formats include PDF, Word, HTML, slideshows, and more. An R Markdown document essentially serves three purposes: communication, collaboration, and as a modern-day lab environment that captures not only what you did, but also what you were thinking. From a communication perspective it enables decision makers to focus more on the results of your analysis rather than the code. However, because it enables you to also include the code, it functions as a means of collaboration between data scientists.

R Markdown uses the rmarkdownpackage, but you don’t have to explicitly load the package in RStudio. RStudio will automatically load the package as needed. The output format of an R Markdown file is a plain text file with an extension of Rmd. These files contain a mixture of three types of content including a YAML header, R code, and text mixed with simple text formatting.

The output R markdown file contains both code and the output of the code. Using the RStudio interface you can run sections of the code or all the code in the file. You can see an example of this in the screenshot below. Notice that the code is enclosed by three back-ticks followed by the output of the code below.

```
3.  Now let's do something a little more interesting. First, use the `dplyr` `mutate()`
function to group the fires by decade.
```{r}
df = mutate(df, DECADE = ifelse(YEAR_ %in% 1980:1989, "1980-1989", ifelse(YEAR_ %in%
1990:1999, "1990-1999", ifelse(YEAR_ %in% 2000:2009, "2000-2009", ifelse(YEAR_ %in%
2010:2016, "2010-2016", "-99")))))
```

4. Next, color code the wildfires by `DECADE` and create a graduated symbol map based on
the size of each fire. The `colour` property defines the column to use for grouping, and the
`size` property defines the column to use for the size of each symbol.
```{r warning=FALSE, error=FALSE, message=FALSE}
ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y = DLATITUDE, colour= DECADE, size =
TOTALACRES))
```
```

If you want to export the contents to a specific file type you can use the Knit functionality embedded in RStudio to export to HTML, PDF, and Word formats. This will export a complete file containing text, code, and results.

In this chapter we’ll cover the following topics:

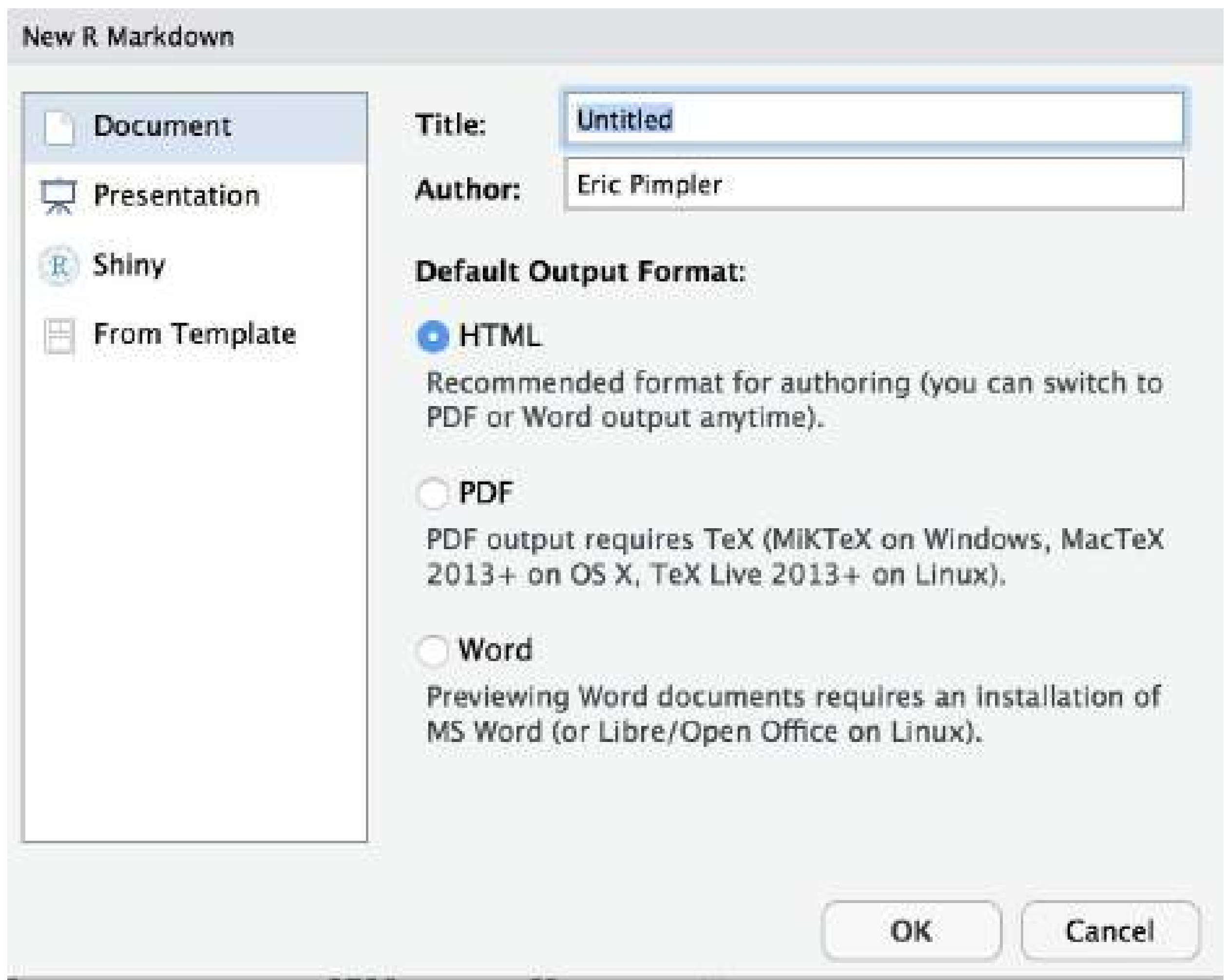
- Creating a R Markdown file
- Adding code chunks and text to an R Markdown file
- Code chunk and header options
- Caching
- Using Knit to output an R Markdown file

Exercise 1: Creating an R Markdown file

An R Markdown file is simply a plain text file with a file extension of .Rmd. You can use RStudio to create new markdown files, which is what you'll do in this brief exercise.

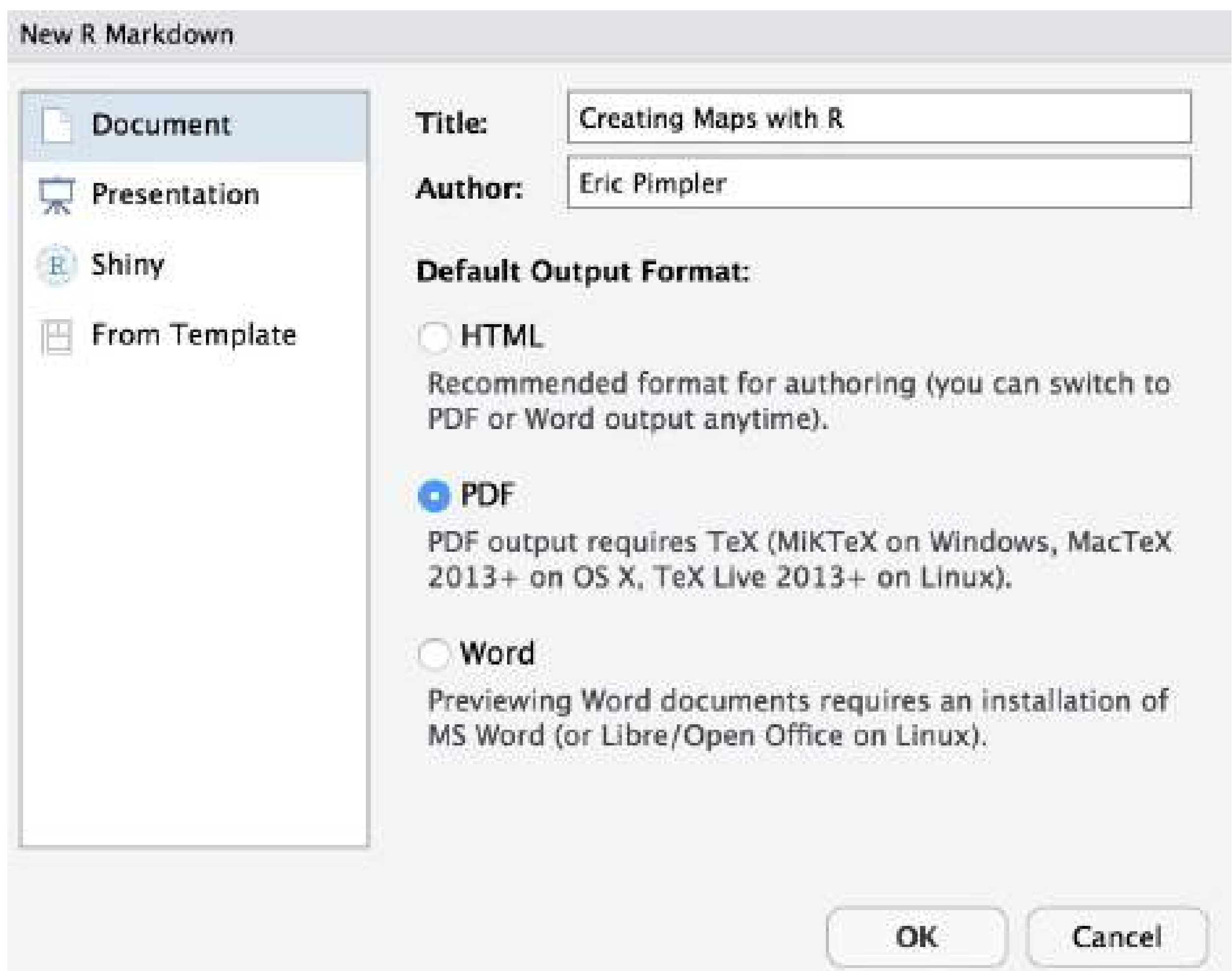
1. The exercises in this chapter require the following packages: readr, dplyr, ggplot2, and ggmap. They can be loaded from the **Packages** pane, the **Console** pane, or a script.

2. Open RStudio and go to File | New File | R Markdown. This will display the dialog you see below. There are different types of markdown that can be created, but for this exercise we'll keep it simple and create a document.



3. Select Document (which is the default), give it a title of Creating Maps with

R, change the author name if you'd like, and select PDF as the output.



4. This will create a file with some header information, text, and code. Your file should look similar to the screenshot below.

```
---
title: "Creating Maps with R"
author: "Eric Pimpler"
date: "7/18/2018"
output: pdf_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML,
PDF, and MS Word documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as
well as the output of any embedded R code chunks within the document. You can embed an R code
chunk like this:

```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the
R code that generated the plot.
```

5. At the very top of the file is the header information, which is surrounded by dashes. We'll add some content to this section in a later exercise, but for now we'll leave it as is.

```
---
title: "Creating Maps with R"
author: "Eric Pimpler"
date: "7/18/2018"
output: pdf_document
---
```

6. Code sections are

grouped through the use of back-ticks as seen in the screenshot below.

```
```{r pressure, echo=FALSE}
plot(pressure)
```
```

7. Plain text and formatted text can be included in a markdown file as well. Text that needs to be formatted must follow a specific syntax. For example, you format text for italics, bold font, headings, links and images. Below is an example of both plain text and text that has been formatted.

```
## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML,
PDF, and MS Word documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as
well as the output of any embedded R code chunks within the document. You can embed an R code
chunk like this:
```

8. Other than the header information we aren't going to use any of the default code or text provided so go ahead and delete everything other than the header.

9. Save the file to your working directory with a name of

CreatingMapsWithR.Rmd.

Exercise 2: Adding Code Chunks and Text to an R Markdown File

R code can be included in the R Markdown file through the use of chunks, which are defined through the use of three back-ticks followed by an r enclosed within curly braces. Inside the curly braces are options that can be included. These options can include TRUE | FALSE parameters for turning various types of messaging on and off.

Chunks define a single task, sort of like a function. They should be self-contained and tightly defined pieces of code. There are three ways to insert chunks into an R Markdown file: Cmd/Ctrl-Alt-I, the **Insert**button on the editor toolbar, and by manually typing the chunk delimiters.

You can also add plain text and formatted text to an R Markdown file. Formatted text has to be defined according to a specific syntax. We'll see various examples of formatted text as we move through this exercise.

In this exercise you'll learn how to add code chunks to an R Markdown file.

1. First, we'll add some descriptive text that will be included in the output R Markup file. Add the text you see below to the file just below the header. If you have a digital copy of the book you can copy and paste rather than typing everything. Notice that the text Step 1: Creating a Basemap has been preceded by two pound signs. ##Step 1: Creating a Basemap. The pound signs are used to define headings. In this case two pound signs would translate to an HTML <h2> tag, which simply defines the size of the text. You'll also notice that some of the words like ggmap and ggplot are surrounded by single quotes. Single quotes are used to define a different style for the word that indicates this word is programmatic code.

The `ggmap` package enables the visualization of spatial data and spatial statistics in a map format using the layered approach of `ggplot2`. This package also includes basemaps that give your visualizations context including Google Maps, Open Street Map, Stamen Maps, and CloudMade maps. In addition, utility functions are provided for accessing various Google services including Geocoding, Distance Matrix, and Directions.

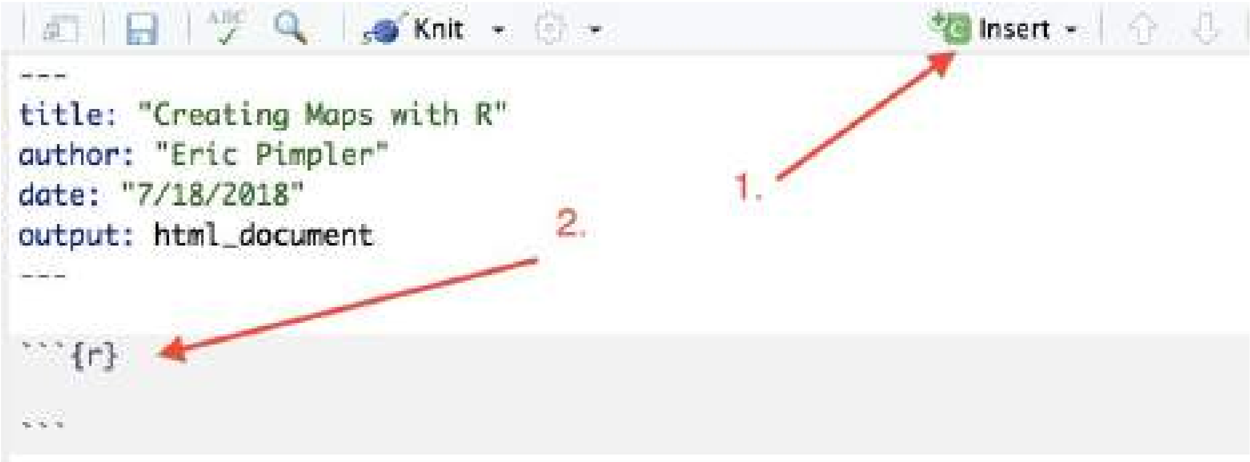
The `ggmap` package is based on `ggplot2`, which means it will take a layered approach and will consist of the same five components found in `ggplot2`. These include a default dataset with aesthetic mappings where x is longitude, y is latitude, and the coordinate system is fixed to Mercator. Other components include one or more layers defined with a geometric object and statistical transformation, a scale for each aesthetic mapping, coordinate system, and facet specification. Because `ggmap` is built on `ggplot2` it has access to the full range of `ggplot2` functionality.
In this exercise you'll learn how to use the `ggmap` package to plot various types of spatial visualizations.

##Step 1: Creating a Basemap
There are two basic steps to create a map with `ggmap`. The details are more complex than these two steps might imply, but in general you just need to download the map raster and then plot operational data on the basemap. Step 1 is

to download the map raster, also known as the basemap. This is accomplished using the `get_map()` function, which can be used to create a basemap from Google, Stamen, Open Street Map, or CloudMade. You'll learn how to do that in this step. In a future step you'll learn how to add and style operational data in various ways.

1. First, load the libraries that we'll need for this exercise

2. Click **Insert** and then **R** to insert a new code chunk as seen below. The code you add will go in between the set of back-ticks. Most markdown files will have a number of code chunks, with each defining a specific task. They are similar in many ways to functions.



3. For this code chunk we'll just load the libraries that will be used in this exercise. Add the code you see below inside the code chunk boundaries.

```{r}
library(ggplot2) library(ggmap) library(readr) library(dplyr) ```

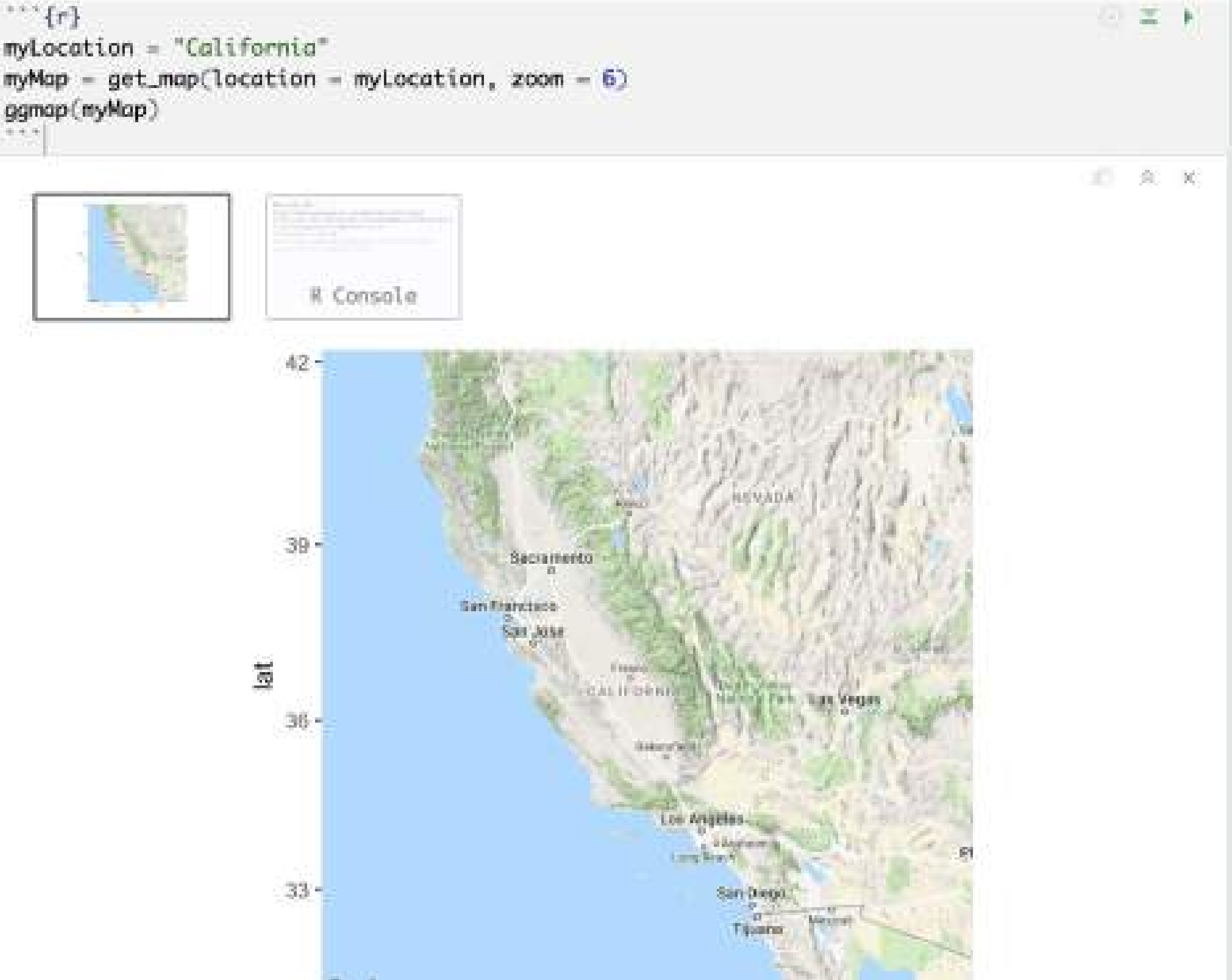
4. Add some additional text that describes the next step.

2. Create a variable called `myLocation` and set it to `California`. Call the `get_map()` function with a zoom level of
6, and plot the map using the `ggmap()` function, passing in a reference to the variable returned by the `get_map()` function. The default map type is Google Maps with a style of Terrain.

5. Insert a new code chunk just below the descriptive text and add the following code.
```

````
```{r}
myLocation <- “California”
myMap <- get_map(location = myLocation, zoom = 6) ggmap(myMap)
````

6. Let’s run the code that has been added so far to see the result. Select Run | Run All from the RStudio interface. This should produce the output you see below. The output is included inside the markdown document. If not, check your code and try running it again.
````



7. Add descriptive text for the next section.
3. The code you see below will create a Google satellite basemap layer. Other basemap layers include Stamen, OSM, and CloudMade.
8. Create a new code chunk and add the code you see below.

~~~~
```{r}
myMap <- get_map(location = myLocation, zoom = 6, source=”google”,
maptype=”satellite”)
ggmap(myMap)
```
~~~~

9. Add descriptive text for the next section.

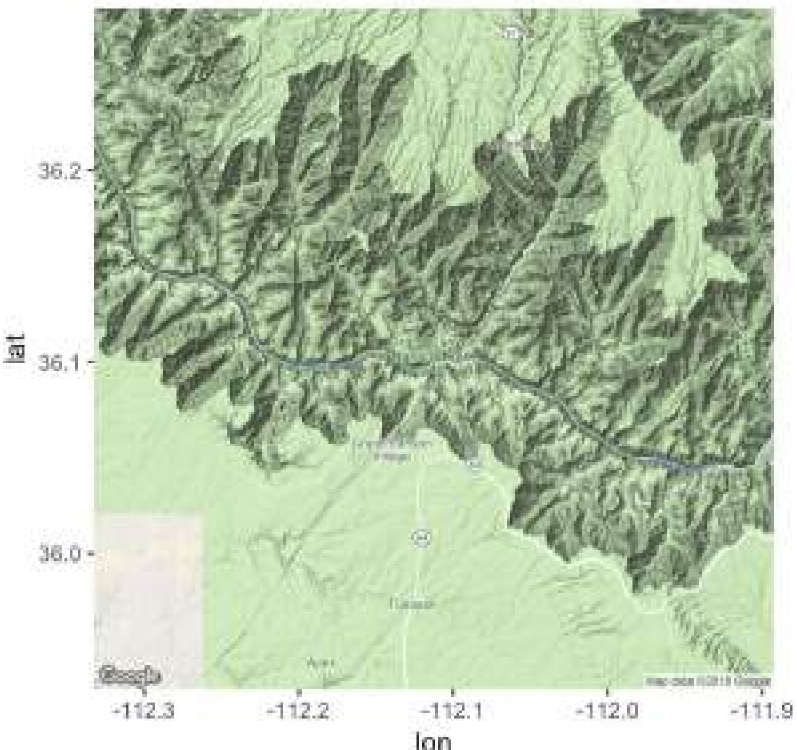
4. There are a number of ways that you can define the input location: longitude/latitude coordinate pair, a character string, or a bounding box. The character string tends to be a more practical solution in many situations since you can simply pass in the name of the location. For example, you could define

the location as Houston Texas or The White House or The Grand Canyon. When a character string is passed to the location parameter it is then passed to the geocoding service to obtain the latitude/ longitude coordinate pair. Add the code you see below to see how passing in a character string works.

10. Create a new code chunk and add the code you see below.

```
```{r}
myMap <- get_map(location = “Grand Canyon, Arizona”, zoom = 11)
ggmap(myMap)
```

11. Let’s stop adding code for now and run what is currently in the file to see the result. Select Run | Run All. Several maps will be produced inside the markup document including the one seen below, which will be produced at the very end. If you don’t see the maps you may need to check your code. We haven’t yet added parameters that will output warnings and errors, but will do so in a later step.
```



12. Add descriptive text for the next section.
The zoom level can be set between 3 and 21 with 3 representing a continent level view, and 21 representing a building level view.

##Step 2: Adding Operational Data Layers
`ggmap()` returns a `ggplot` object, meaning that it acts as a base layer in the `ggplot2` framework. This allows for the full range of `ggplot2` capabilities meaning that you can plot points on the map, add contours and 2D heat maps, and more. We'll examine some of these capabilities in this section.

1. For this section we'll use the historical wildfire information found in the StudyArea_SmallFile.csv file. Load this dataset using the `read_csv()` function. You can download this file at: https://
www.dropbox.com/s/9ouh21a6ym62nsl/StudyArea.csv?dl=0

13. Create a new code chunk and add the code you see below. This will load wildfire data from a csv file. Note: The path to your StudyArea_SmallFile. csv

file may differ from the one you see below.

```
```{r}
dfWildfires <- read_csv("~/Desktop/IntroR/Data/StudyArea_SmallFile. csv",
col_types = list(FIRENUMBER = col_character(), UNIT = col_ character()),
col_names = TRUE)
```
```

14. Add descriptive text for the next section.

2. Initially we'll just load the wildfire events as points. Add the code you see below to produce a map of California that displays wildfires from the years 1980-2016 that burned more than 1,000 acres.

15. Create a new code chunk and add the code you see below. This code chunk will display each of the wildfires as a point on the map.

```
```{r}
myLocation <- 'California'
#get the basemap
myMap <- get_map(location = myLocation, zoom = 6)
# use the select() function to limit the columns from the data frame
df <- select(dfWildfires, STATE, YEAR_, TOTALACRES, DLATITUDE,
DLONGITUDE)
#use the filter() function to get only fires in California with acres
#burned greater than 1000
df <- filter(df, TOTALACRES >= 1000 & STATE == 'California') #produce the
final map
ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y =
DLATITUDE))
```
```

16. Add the following descriptive text.
3. Now let's do something a little more interesting. First, use the `dplyr` `mutate()` function to group the fires by decade.

17. Create a new code chunk and add the code you see below. The mutate() function is used in this code chunk to create a new column called DECADE and then populate each row with a value for the decade in which the fire occurred.

```{r}

```{r}
1989”, ifelse(YEAR_ %in% 1990:1999, “1990-1999”, ifelse(YEAR_ %in%
2000:2009, “2000-2009”, ifelse(YEAR_ %in% 2010:2016, “2010-2016”,
“-99”)))))
```

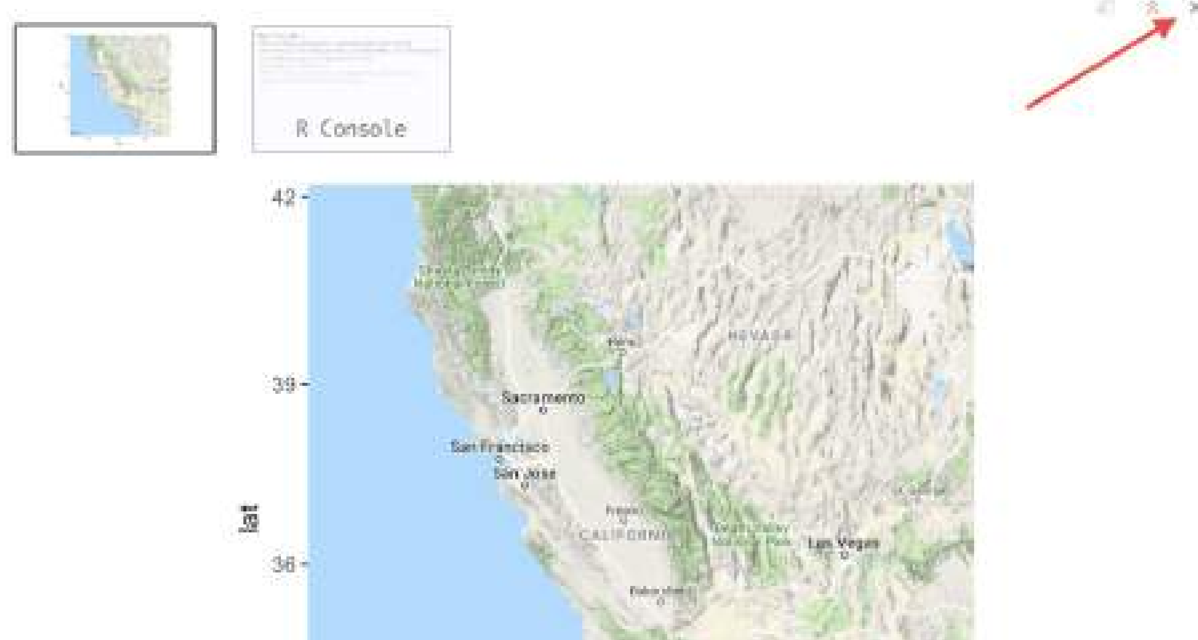
18. Add the following descriptive text.

4. Next, color code the wildfires by `DECADE` and create a graduated symbol map based on the size of each fire. The `colour` property defines the column to use for grouping, and the `size` property defines the column to use for the size of each symbol.

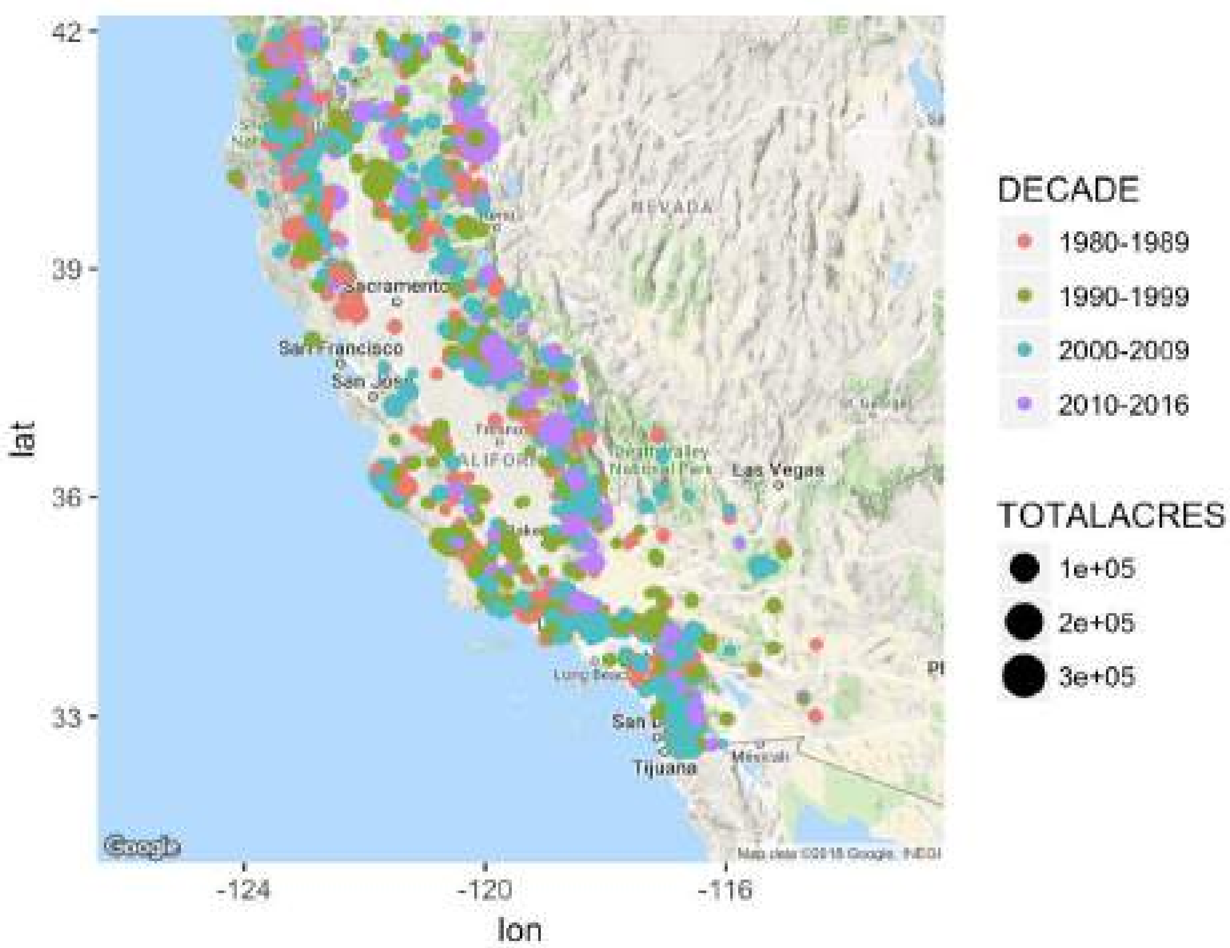
19. Create a new code chunk and add the code you see below. This code chunk will color code the fires by decade.

```{r}
ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y =
DLATITUDE, colour= DECADE, size = TOTALACRES))

20. Let’s stop adding code for now and run what is currently in the file to see the result. Before running the code again go ahead and clear the past results by clicking the small X in the upper right hands corner of the output for each map as seen in the screenshot below.




21. Select Run | Run All. The output produced will include several maps with the final map appearing as seen in the screenshot below.




5. Let's change the map view to focus more on southern California, and in particular the area just north of Los Angeles.

23. Create a new code chunk and add the code you see below. This code chunk will color code the fires by decade and size the symbols according the total acreage burned.

````
```{r}
myMap <- get_map(location = "Santa Clarita, California", zoom = 10)
ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y =
DLATITUDE, colour= DECADE, size = TOTALACRES))
```
````

24. Add the following descriptive text.

6. Next we'll add contour and heat layers. The `geom_density2d()` function is used to create the contours while the `stat_ density2d()` function creates the heat

map. Add the following code to produce the map you see below. You can experiment with the colors using the `scale_fill_gradient(low and high)` properties. Here we've set them to green and red respectively, but you may want to change the color scheme.

25. Create a new code chunk and add the code you see below. This code chunk will create a heat map and add contours.

```{r}
myMap <- get_map(location = "Santa Clarita, California", zoom = 8)

ggmap(myMap, extent = "device") + geom_density2d(data = df, aes(x = DLONGITUDE, y = DLATITUDE), size = 0.3) + stat_density2d(data = df, aes(x = DLONGITUDE, y = DLATITUDE, fill = ..level.., alpha = ..level..), size = 0.01, bins = 16, geom = "polygon") + scale_fill_ gradient(low = "green", high = "red") + scale_alpha(range = c(0, 0.3), guide = FALSE)
```

7. If you'd prefer to see the heat map without contours, the code can be simplified as follows:

27. Create a new code chunk and add the code you see below. This code chunk will remove the contours.

```{r}
ggmap(myMap, extent = "device") + stat_density2d(data = df, aes(x = DLONGITUDE, y = DLATITUDE, fill = ..level.., alpha = ..level..), size = 0.01,bins = 16, geom = "polygon") + scale_fill_gradient(low = "green", high = "red") + scale_alpha(range = c(0, 0.3), guide = FALSE)
```

28. Add the following descriptive text.

8. Finally, let's create a facet map that depicts hot spots for each year in the current decade. Add the following code to see how this works. The dataset contains information up through the year 2016.

29. Create a code chunk and add the code you see below.

```{r}
df <- filter(dfWildfires, STATE == 'California')
df <- filter(df, YEAR_ %in% c(2010, 2011, 2012, 2013, 2014, 2015, 2016))
myMap <- get_map(location = "Santa Clarita, California", zoom = 9)
```

ggmap(myMap, extent = “device”) + stat_density2d(data = df, aes(x = DLONGITUDE, y = DLATITUDE, fill = ..level.., alpha = ..level..), size = 0.01,bins = 16, geom = “polygon”) + scale_fill_gradient(low = “green”, high = “red”) + scale_alpha(range = c(0, 0.3), guide = FALSE) + facet_wrap(~ YEAR_)

30. That completes the code for this R Markdown file. Go ahead and run the code again to see the final output by selecting
Run | Run All.

Exercise 3: Code chunk and header options

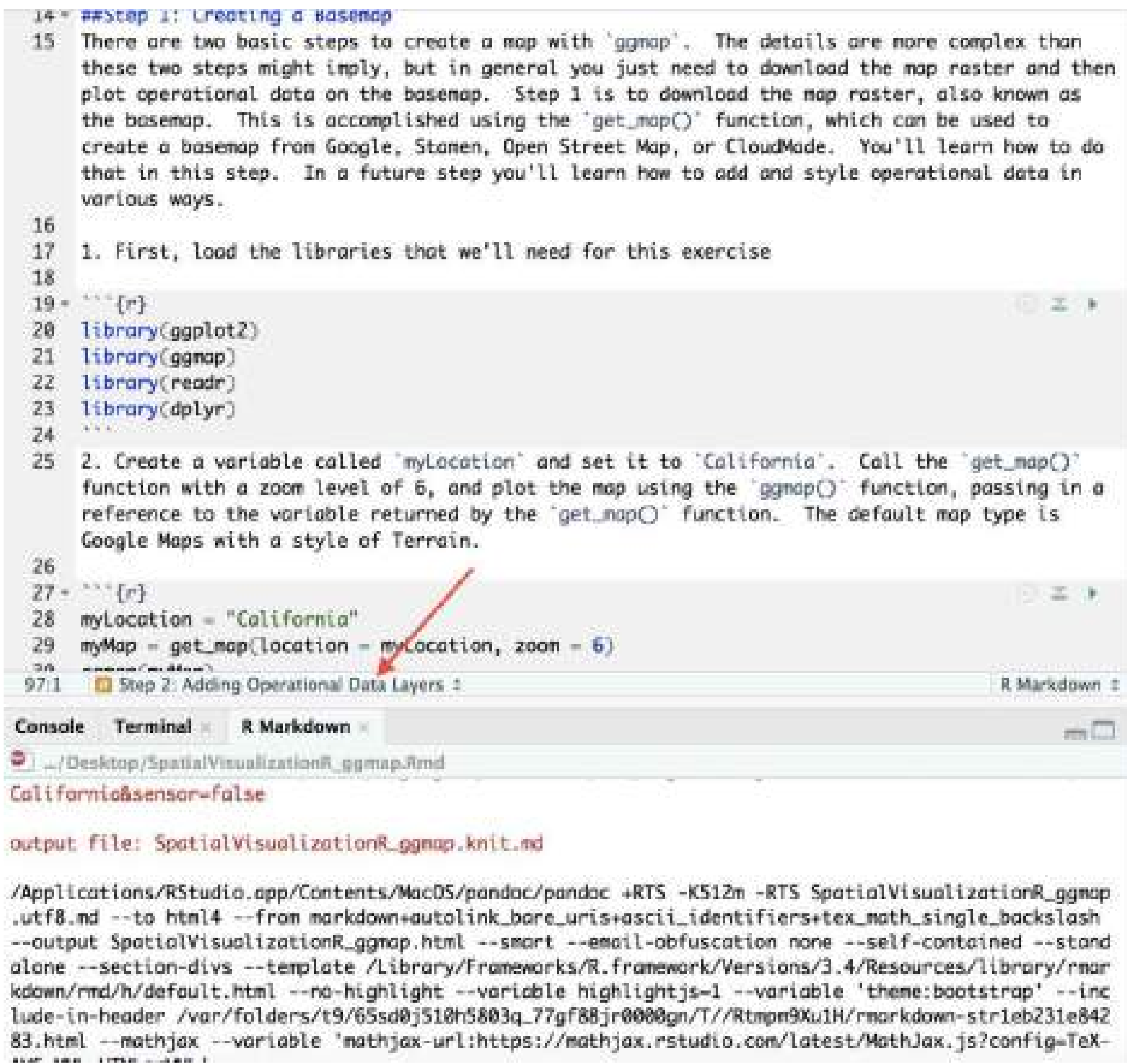
Chunk options are arguments supplied to the chunk header. Currently there are approximately 60 such options. We’ll examine some of the more commonly used and important options in this exercise. All code chunk options are placed inside the {r} block.

Code chunks can be given an optional name as seen in the example code below where the code chunk has been given a name of
MapSetup
.

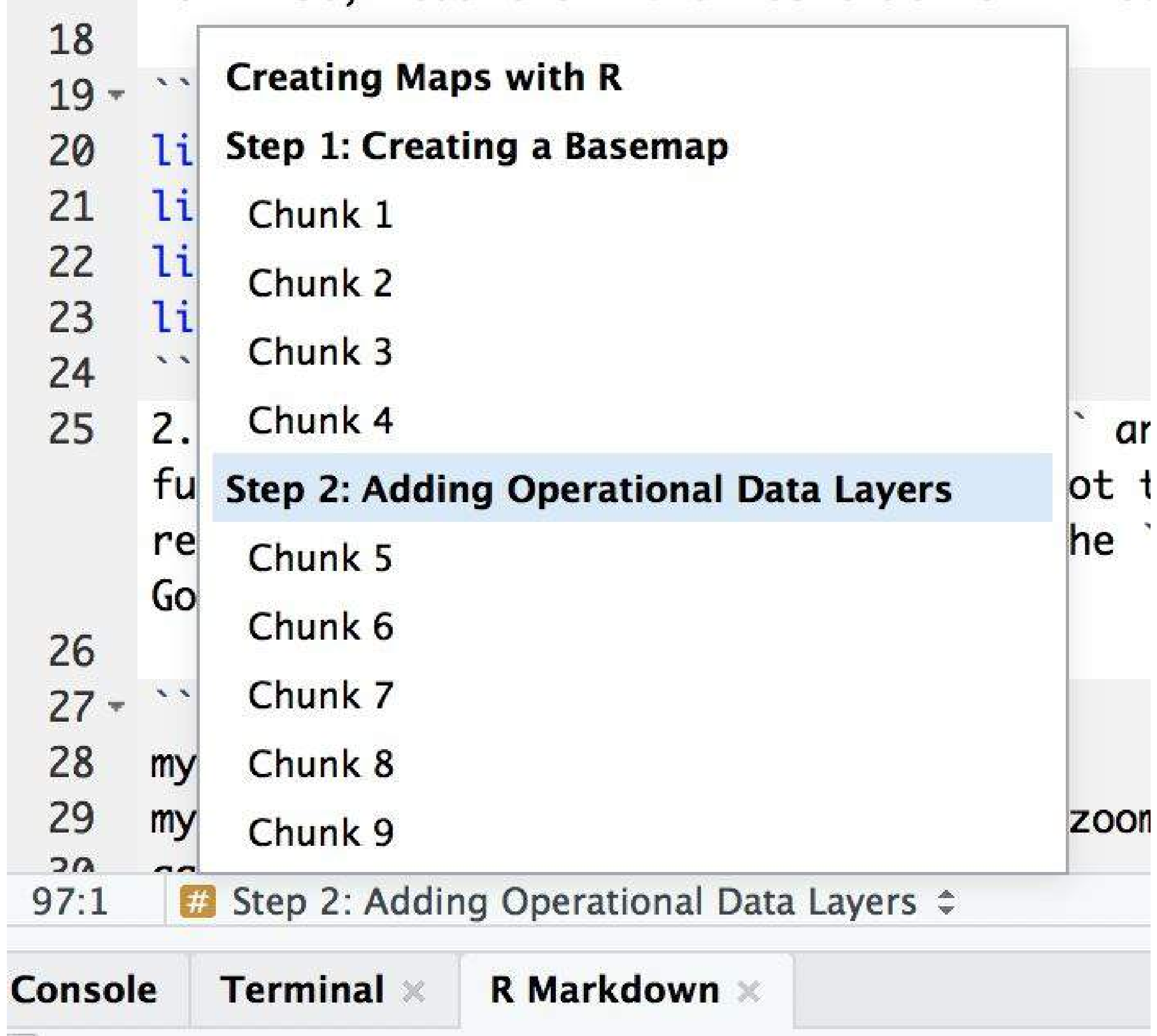
```
```{r MapSetup, warning=FALSE, error=FALSE, message=FALSE}
```

The advantages of naming chunks include easier navigation using the code navigator in RStudio, useful names given to graphics produced by chunks, and the ability to cache chunks to avoid re-performing computations on each run. This last advantage is perhaps the most useful.
1. The R Markdown pane includes a quick access menu for easily navigating

to different sections of your R Markdown page. The arrow in the screenshot below displays the location of this functionality.
```



2. Click on the quick access button now to see the different sections of the R Markdown file. You should see something similar to the screenshot below. You'll notice that it is sectioned by headings and then code chunks. To make navigation easier you can name each of these chunks.



Select Chunk 1 under Step 1: Creating a Basemap to return to the first code chunk you created in an earlier exercise. This code chunk simply defines the libraries that will be used in the file.

In the {r} section of the header name the chunk libs.

```{r libs}

3. Notice that the value has now been updated in the quick access dropdown menu.
```

Creating Maps with R

Step 1: Creating a Basemap

Chunk 1: libs

Chunk 2

Chunk 3

Chunk 4

Step 2: Adding Operational Data Layers

Chunk 5

Chunk 6

Chunk 7

Chunk 8

Chunk 9

4. Rename the rest of your code chunks. You can use whatever name makes the most sense for each.

5. Next, we'll add some code options. Although there are currently 60+ options that can be applied to a code chunk we'll examine only a few of the more important options. You can get a list of all the available code chunk options at https://www.rstudio.com/wp-content/uploads/2015/03/ rmarkdown-reference.pdf.

6. Messaging is one of the most commonly used and useful options. There are actually three messaging options: messages, warnings, errors. All three are TRUE | FALSE values that can be set and all are set to FALSE by default. Navigate to Chunk 2 and add the options you see highlighted below. This will turn on the messaging for any general information messages, warnings, and errors.

```{r **error=TRUE, warning=TRUE, message=TRUE**}
```

```
myLocation <- “California”
myMap <- get_map(location = myLocation, zoom = 6)
ggmap(myMap)
```

7. Now when you run this section any of these messages will be printed out along with the output. Rather than running the entire markdown file code each time you want to test something you can limit the run to a particular code chunk by clicking the arrow on the far-right hand side of the code chunk as seen in the screenshot below.

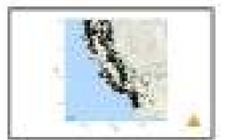
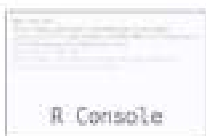
```
```{r error=TRUE, warning=TRUE, message=TRUE}
myLocation = "California"
myMap = get_map(location = myLocation, zoom = 6)
ggmap(myMap)
```
```

R Console

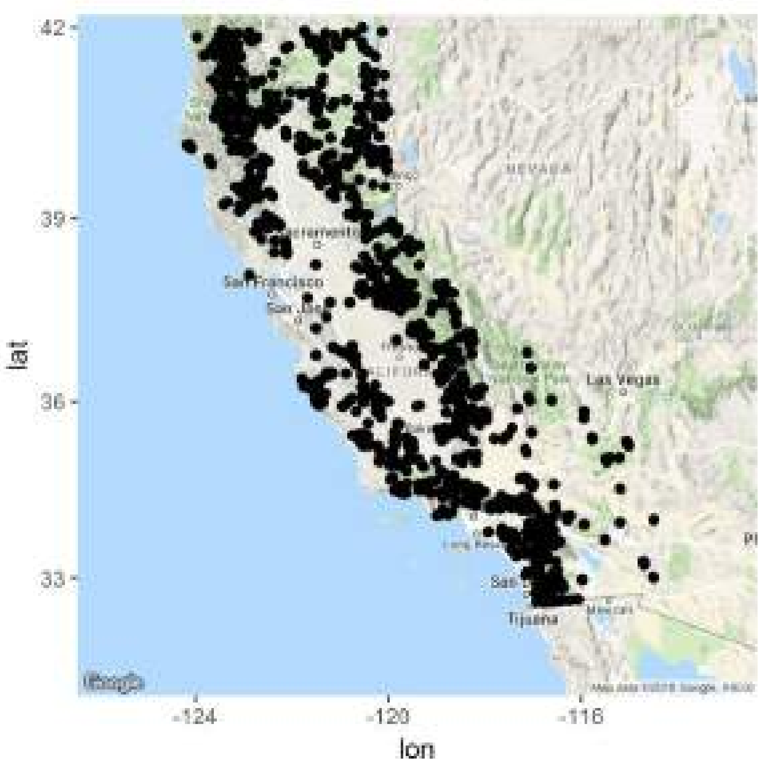
8. The output window includes two overview windows: the output visualization and the R Console. If you click the R Console overview window as seen in the screenshot below it will display any messages that were written to the console as a result of the execution of this code block.

Clicking the R Console window should produce an output similar to the screenshot below.

9. Now add the same message, warning, and error options to your other code chunks.
10. Run the code chunks one at a time an examine the output. Any warning and errors will be prominently displayed as seen in the screenshot below.

Removed 5 rows containing missing values (geom_point).



11. You can also define document wide options as well. In this step we'll look at a common option defined in the header. The content of the header defines parameters that control various settings for the entire document.

The header can include basic descriptive information including the title, author, date, and output format along with other settings including parameters and bibliographies and citations.
Parameters are used when you need to re-render the same report but with distinct values for inputs. The params field controls these parameters.

You'll notice in the code example below that a state parameter has been defined with a value of California. This value can then be accessed elsewhere in the R Markdown file using the syntax params$<parameter> or params$state in this example.

```
---
title: "Creating Maps with R"
author: "Eric Pimpler"
date: "7/18/2018"
output: html_document
params:
  state: 'California'
---
```

Add the params options with a parameter of state and set it equal to California in your file exactly as seen in the screenshot above.

12. Navigate to Chunk 2 and find the line you see below.

myLocation <- “California”

13. Change this line as seen below to access the state parameter.

```{r error=TRUE, warning=TRUE, message=TRUE}
**myLocation <- params$state**
myMap <- get_map(location = myLocation, zoom = 6) ggmap(myMap)
```

14. Run the code for Chunk 2 only and you should see the same output map centered on California.

15. Clear the output for chunk 2 by clicking the X in the upper right-hand corner of the output.

16. Return to the state parameter in the header and change the value to Montana.

--
title: “Creating Maps with R”
author: “Eric Pimpler”

date: “7/18/2018”
output: html_document
params:

state: ‘Montana’

17. Run code chunk 2 again and now the map should be centered on Montana.

Exercise 4: Caching

Code chunks can also be cached, which is great for computation that takes a long time to execute. To enable caching the cache parameter should be set to TRUE. This will save the output of the code chunk to a specially named file on desk. On any subsequent runs, knitr checks to see if the code has changed, and if not, it will reuse the cached results.

You do need to be careful with caching though as it will only re-run a code chunk if the code changes. However, it doesn't take into account things such as changes to underlying data sources. For example, the data in an underlying data source could change, but because the R Markdown file will only re-run the code chunk if the code changes, this could become an issue.

1. Find the code chunk you see below that maps the individual wildfire points. You may have named the chunk something other than what I have named the chunk (point_map).

```
```{r point_map, error=TRUE, warning=TRUE, message=TRUE} myLocation
<- 'California'
#get the basemap
myMap <- get_map(location = myLocation, zoom = 6)
# use the select() function to limit the columns from the data frame
df <- select(dfWildfires, STATE, YEAR_, TOTALACRES, DLATITUDE,
DLONGITUDE)
#use the filter() function to get only fires in California with acres
#burned greater than 1000
df <- filter(df, TOTALACRES >= 1000 & STATE == 'California') #produce the
final map
ggmap(myMap) + geom_point(data=df, aes(x = DLONGITUDE, y =
DLATITUDE))
```
```

2. Add the cache parameter to the options for the chunk as seen below.
```{r point_map, **cache=TRUE**, error=TRUE, warning=TRUE,

3. This code chunk is dependent upon the data in the dfWildfires data frame, which is loaded in the code chunk directly preceding this chunk. The code chunk that loads the data from a csv file into the dfWidlfires variable can be seen below. You may have named the chunk differently (load_data).

```
```{r load_data, error=TRUE, warning=TRUE, message=TRUE} dfWildfires <-
read_csv("~/Desktop/IntroR/Data/StudyArea_SmallFile. csv", col_types =
list(FIRENUMBER = col_character(), UNIT = col_ character()), col_names =
TRUE)
```
```

4. Because the point_map code chunk is dependent upon the data in the dfWildfires data frame you need to add a dependson parameter to the point_map code chunk.

```{r point_map, cache=TRUE, **dependson='load_data'**, error=TRUE, warning=TRUE, message=TRUE}

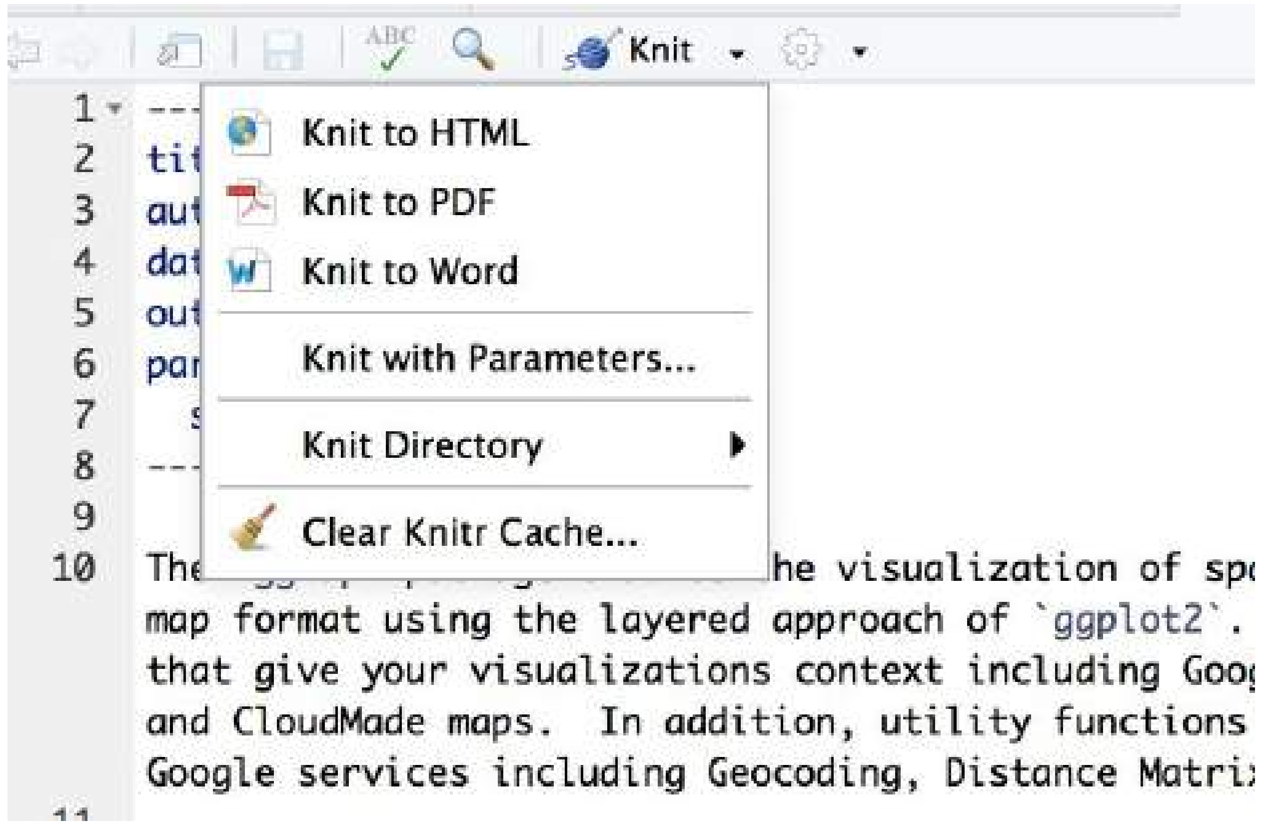
This will cover situations where the read_csv() call changes. For example, a different file might be read by the function.

5. Keep in mind that the cache and dependson parameters only monitor for changes in the .Rmd file. What would happen if the underlying data in the StudyArea_SmallFile.csv file changes? The answer is that the changes wouldn't be picked up. To handle this sort of situation you can use the cache.extra option along with the file.info() function.

```{r load_data, **cache.extra = file.info('~/Desktop/IntroR/ Data/StudyArea_SmallFile.csv')** error=TRUE, warning=TRUE,

## Exercise 5: Using Knit to output an R Markdown file

The Knitfunctionality built into RStudio can be used to export an R Markdown file to various formats including HTML, PDF, and Word. Knit can be accessed from the dropdown menu seen in the screenshot below.




1. To simplify the output of the R Markdown file you're going to remove some of the options that were added in previous exercise. In the
```

CreateMapsWithR.rmd file remove cache, dependson, and cache. extra parameters added in the last exercise.

2. Select Knit and find the Knit Directory menu item from the RStudio interface. By default, it is set to Document Directory. This simply means that the output file will go into the same directory where the R Markdown file has been saved.

3. Select Knit | Knit to HTML. Knit will begin processing the file and you’ll see output messaging information written to the **Console** pane. If everything goes as expected an output HTML file called CreatingMapsWithR.html will be created in the same folder where the CreatingMapsWithR.Rmd file was saved. The output file will be fairly length, but the top part should look similar to the screenshot below.

Creating Maps with R

Eric Pimpler

7/18/2018

The `ggmap` package enables the visualization of spatial data and spatial statistics in a map format using the layered approach of `ggplot2`. This package also includes basemaps that give your visualizations context including Google Maps, Open Street Map, Stamen Maps, and CloudMade maps. In addition, utility functions are provided for accessing various Google services including Geocoding, Distance Matrix, and Directions.

The `ggmap` package is based on `ggplot2`, which means it will take a layered approach and will consist of the same five components found in `ggplot2`. These include a default dataset with aesthetic mappings where x is longitude, y is latitude, and the coordinate system is fixed to Mercator. Other components include one or more layers defined with a geometric object and statistical transformation, a scale for each aesthetic mapping, coordinate system, and facet specification. Because `ggmap` is built on `ggplot2` it has access to the full range of `ggplot2` functionality.

In this exercise you'll learn how to use the `ggmap` package to plot various types of spatial visualizations.

Step 1: Creating a Basemap

There are two basic steps to create a map with `ggmap`. The details are more complex than these two steps might imply, but in general you just need to download the map raster and then plot operational data on the basemap. Step 1 is to download the map raster, also known as the basemap. This is accomplished using the `get_map()` function, which can be used to create a basemap from Google, Stamen, Open Street Map, or CloudMade. You'll learn how to do that in this step. In a future step you'll learn how to add and style operational data in various ways.

1. First, load the libraries that we'll need for this exercise

```
library(ggplot2)
library(ggmap)
library(readr)
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

4. You can check your work against the CreatingMapsWithR.Rmd solution file.

Conclusion

In this chapter you learned how to create an R Markdown file, which can be used to share your work with others in various formats including PDF, Word, HTML, slideshows, and more. R Markdown files can include code, results, and commentary, making them a perfect resource for explaining not only the results of a project, but also the mechanics of how the work was accomplished.

In the next chapter you'll tackle a case study that examines wildfire activity in the western United States.

Chapter 10

Case Study – Wildfire Activity in the Western United States

Studies suggest that over the past few decades, the number and size of wildfires have increased throughout the western United States. The average length of wildfire season has increased significantly as well in some areas. According to the Union of Concerned Scientists (UCS), every state in the western US has experienced an increase in the average annual number of large wildfires (greater than 1,000 acres) over the past few decades. The Pacific Northwest, including Washington, Oregon, Idaho, and the western half of Montana have had particularly challenging wildfire seasons in recent years.

The 2017 wildfire season shattered records and cost the U.S. Forest Service an unprecedented $2 billion. From the Oregon wildfires to late season fires in Montana, and the highly unusual timing of the California fires in December, it was a busy year in the western United States. While 2017 was a particularly notable wildfire season, this trend is nothing new and research suggests we can expect this unfortunate trend to continue due to climate change and other factors. A recent study suggests that over the next two decades, as many as 11 states are predicted to see the average annual area burned increase by 500 percent.

Extensive studies have found that large forest fires in the western US have been occurring nearly five times more often since the 1970s and 80s. Such fires are burning more than six times the land area as before and lasting almost five times longer.

Climate change is thought to be the primary cause of the increase in large wildfires with rising temperatures leading to earlier and decreased volume of snow melts, decreased precipitation, and forest conditions that are drier for longer periods of time. An increase in forest tree disease from insect disturbance has also been associated with climate change and can lead to large areas of highly flammable dead or dying forests. Other potential causes of increased wildfire activity include forest management practices, and an increase in human caused wildfires due to accidents or arson.

In this case study you will use the skills you have gained in this book along with wildfire data from the Federal Wildland Fire Occurrence Database,

(https://wildfire.cr.usgs.gov/ firehistory/data.html), provided by the U.S. Geological Survey (USGS) to visualize the change in wildfire activity from 1980 to 2016. Analysis will be limited to the western United States including California, Arizona, New Mexico, Colorado, Utah, Nevada, Utah, Oregon, Washington, Idaho, Montana, and Wyoming. We were particularly interested in the surge of large wildland fires, categorized as fires that burn greater than 1,000 acres.

So, has wildfire activity and size actually increased, or does it just seem that way because we're tuned in more to bad news and social media? In this chapter you'll answer those questions and more using R with the tidyverse package.

In this chapter we'll answer the following questions:

- Have the number of wildfires increased or decreased in the past few decades?
- Has the acreage burned increased over time?
- Is the size of individual wildfires increasing over time?
- Has the length of the fire season increased over time?
- Does the acreage burned differ by federal organization?

Exercise 1: Have the number of wildfires increased or decreased in the past few decades?

The StudyArea.csvfile in your IntroR\Data folder contains all non-prescribed wildfire activity from 1980-2016 for the 11 states in our study area, which include California, Oregon, Washington, Idaho, Nevada, Arizona, Utah, Montana, Wyoming, Colorado, and New Mexico. We'll use this file for all the exercises in this chapter. We're going to focus primarily on large wildfires in this study, defined here as any non-prescribed fire greater than 1,000 acres.

1. In your IntroR folder create a new folder called CaseStudy1. You can do this inside RStudio by going to the **Files** pane and selecting **New Folder** inside your working directory.

2. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise1.R
. 3. At the top of the script, load the packages that will be used in this exercise.

library(readr) library(dplyr) library(ggplot2)

4. Use the
read_csv() function from the readr package to load the data into a data frame.
df <- read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
5. Check the number of rows in the data frame. This should return 439362 or something close to that.
nrow(df) [1] 439362
6. We only need a few of the columns from the data frame for this exercise so

use the select() function to retrieve the STATE, YEAR_, TOTALACRES, and CAUSE columns. We'll also rename some of these columns in this step. Piping will be used for the rest of the code in this exercise so begin the statement as seen below.

```
df %>%
select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
```

7. Next, filter the data frame so that only wildfires that burned 1,000 acres or more are included. Add the code highlighted in bold below.

```
df %>%
select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
```

8. Group the records by year.

```
df %>%
select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(YR) %>%
```

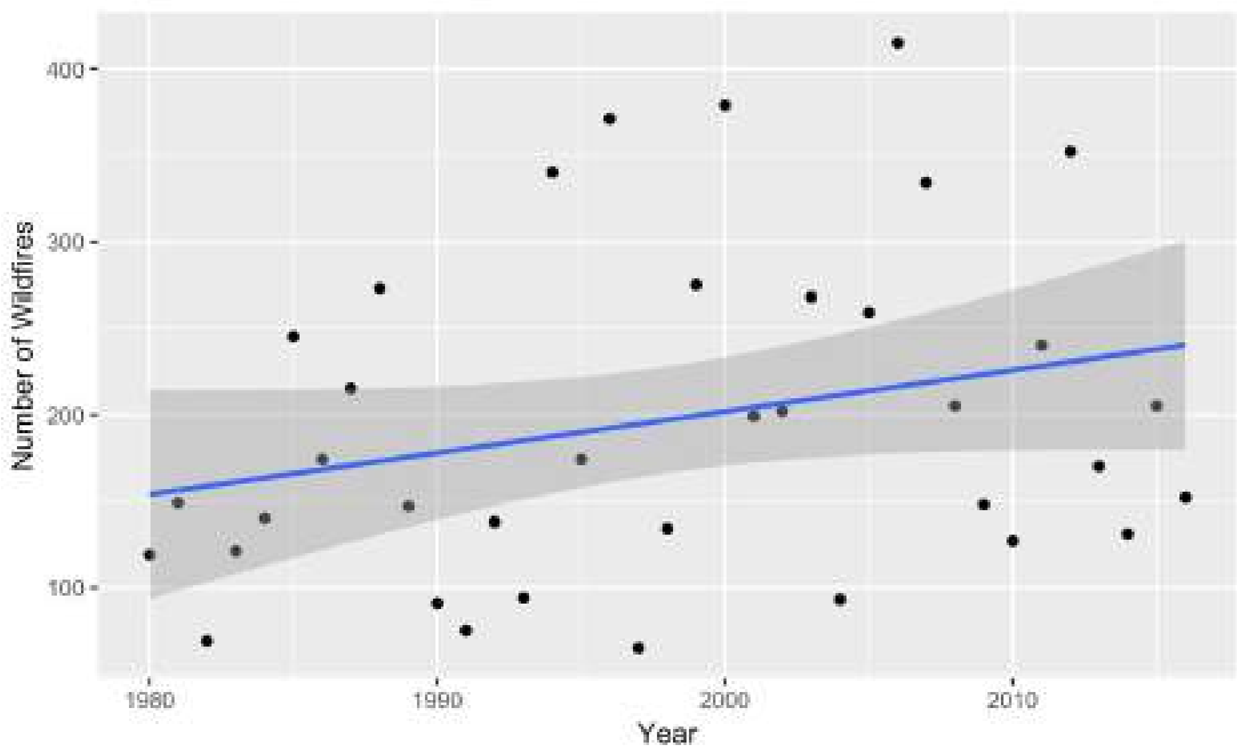
9. Get a count of the number of wildfires for each year by using the summarize() function with the count=n()
parameter.

```
df %>%
select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(YR) %>%
summarize(count=n()) %>%
```

10. Finally, create a scatterplot with a regression line that depicts the number of wildfires over the years.

```
df %>%
select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(YR) %>%
summarize(count=n()) %>%
ggplot(mapping = aes(x=YR, y=count)) + geom_point() +
geom_smooth(method=lm, se=TRUE) + ggtitle("Large Fires Are Becoming
More Common in the West - 1980-2016") + xlab("Year") + ylab("Number of
Wildfires")
```

11. You can check your work against the solution file CS1_Exercise1.R. 12. Save the script and then click the Run button. If you've coded everything correctly you should see the plot displayed in the screenshot below.



13. Based on this visualization it appears as though large wildfires have indeed become more common over the past few decades. But let's expand this to see if all the states in the study area have the same pattern. 14. Create a new R script

and save it with a name of CS1_Exercise1B.R.

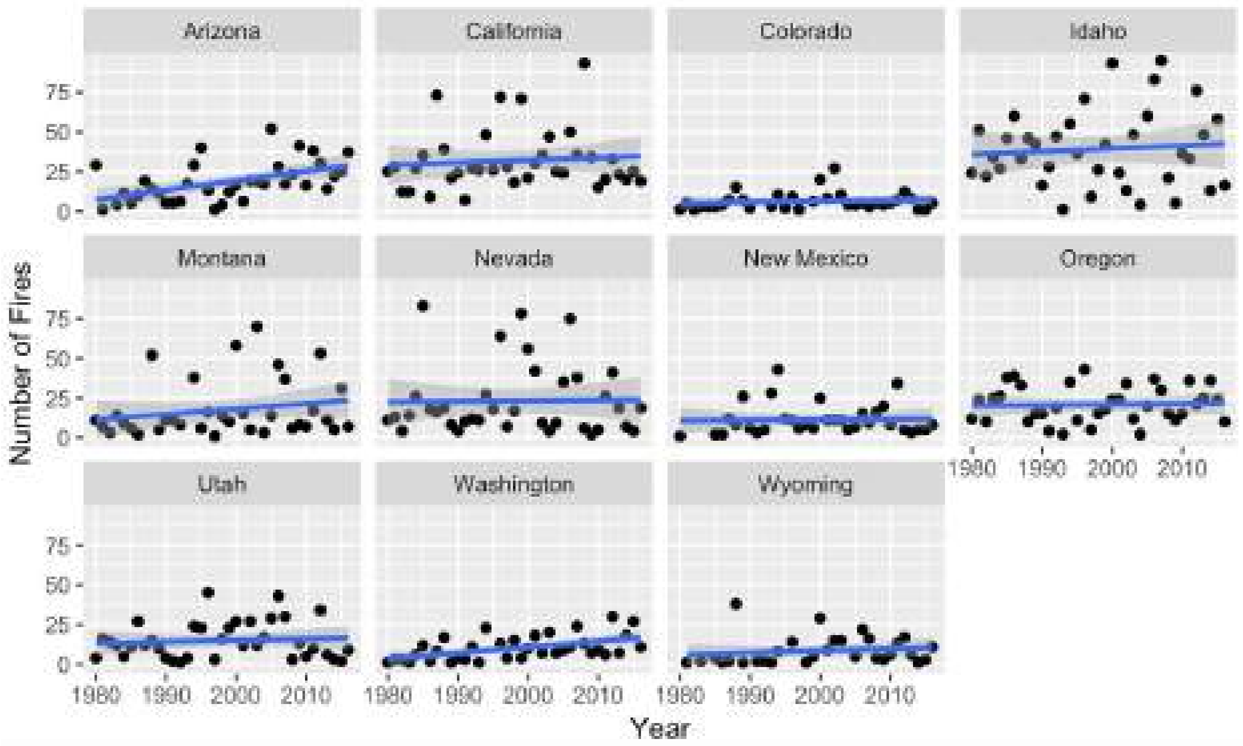
15. Add the following code to your script and save it. We'll discuss the differences between this script and the previous afterward.

```
library(readr) library(dplyr) library(ggplot2)

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(STATE, YR) %>%
summarize(cnt = n()) %>%
ggplot(mapping = aes(x=YR, y=cnt)) + geom_point()+ facet_ wrap(~STATE) +
geom_smooth(method=lm, se=TRUE) + ggtitle("Number of Fires by State and
Year") + xlab("Year") + ylab("Number of Fires")
```

16. Save the script and then run it to see the output shown in the screenshot below.



This script groups the dataset by STATE and YR and then summarizes the data by generating a count of the number for this grouping. Finally, the facet_ wrap() function is used with ggplot() to create the facet map that depicts the number of fires by state over time. A number of the individual states show a slight upward trend over time, but many have an almost flat regression line.

17. You can check your work against the solution file CS1_Exercise1B.R
.

18. Challenge 1: Repeat this process to see the results for wildfires greater than 5,000 acres, 25,000 acres, and 100,000 acres. Are these finding consistent with the results of wildfires greater than 1,000 acres?

19. Challenge 2: Repeat the process but this time group the data by year and by wildfires that are naturally occurring. The CAUSE column includes a value of Natural that can be used to group the data. You'll need a compound grouping statement.

Exercise 2: Has the acreage burned increased over time?

Measuring the number of fires over time only tells part of the story. The amount of acreage burned during that time may give us more insight into the patterns in wildfire activity. In this exercise we'll create visualizations that illustrate how much acreage is being burned each year as a result of wildfires.

1. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise2.R.
2. At the top of the script, load the packages that will be used in this exercise.

library(readr) library(dplyr) library(ggplot2)

3. The first few lines of this script will be similar to the previous exercises, so I won't discuss the details of each line. By now you should be able to determine what each of these lines will accomplish anyway. Add the lines shown below.

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
4. Group the data by year.

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(YR) %>%

5. Use the
summarize()
function to sum the total acreage burned by year.

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(YR) %>%
summarize(totalacres = sum(ACRES)) %>%

6. Create a scatterplot with regression line that displays the total acreage burned by year. In this case you'll convert the total acres burned to a logarithmic scale as well.

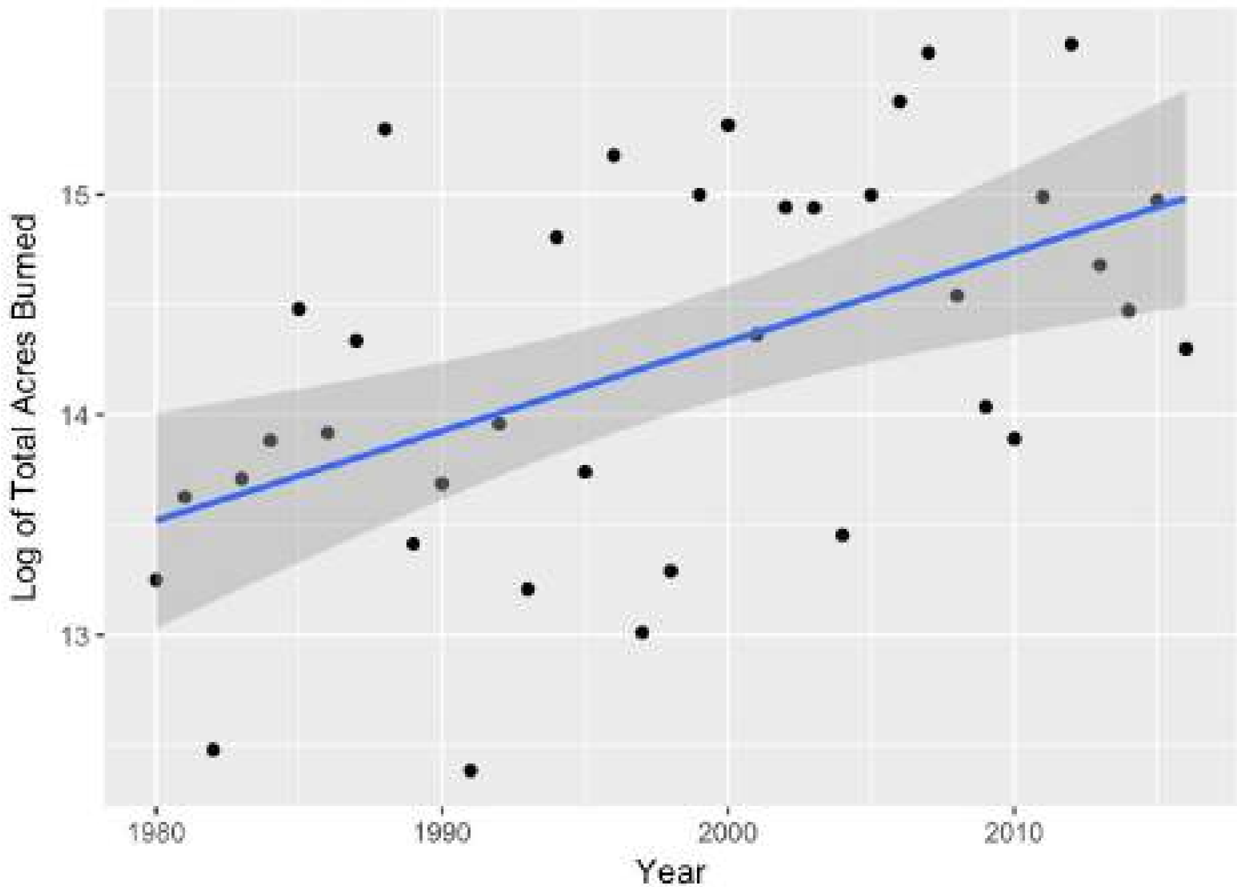
```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(YR) %>%
summarize(totalacres = sum(ACRES)) %>%

ggplot(mapping = aes(x=YR, y=log(totalacres))) + geom_point() +
geom_smooth(method=lm, se=TRUE) + ggtitle("Total Acres Burned") +
xlab("Year") + ylab("Log of Total Acres Burned")
```

7. You can check your work against the solution file CS1_Exercise2.R
.

8. Save the script and then run it to see the output shown in the screenshot below. It's clear from this graph that there has been a significant increase in the acreage burned over the past few decades.



9. Now let’s see if this trend is significant for all states in the study area. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise2B.R.

10. At the top of the script, load the packages that will be used in this exercise.

```
library(readr)
library(dplyr)
library(ggplot2)
```

11. The first few lines of this script will be similar to the previous exercises, so I won’t discuss the details of each line. Add the lines shown below.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
```

filter(ACRES >= 1000) %>%
12. Group the data by
STATE and YR
.

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(STATE, YR) %>%

13. Use the
summarize()
function to calculate the total acreage burned by state.

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(STATE, YR) %>%
summarize(totalacres = sum(ACRES)) %>%

14. Create a facet plot that displays the total acreage burned by state and year.

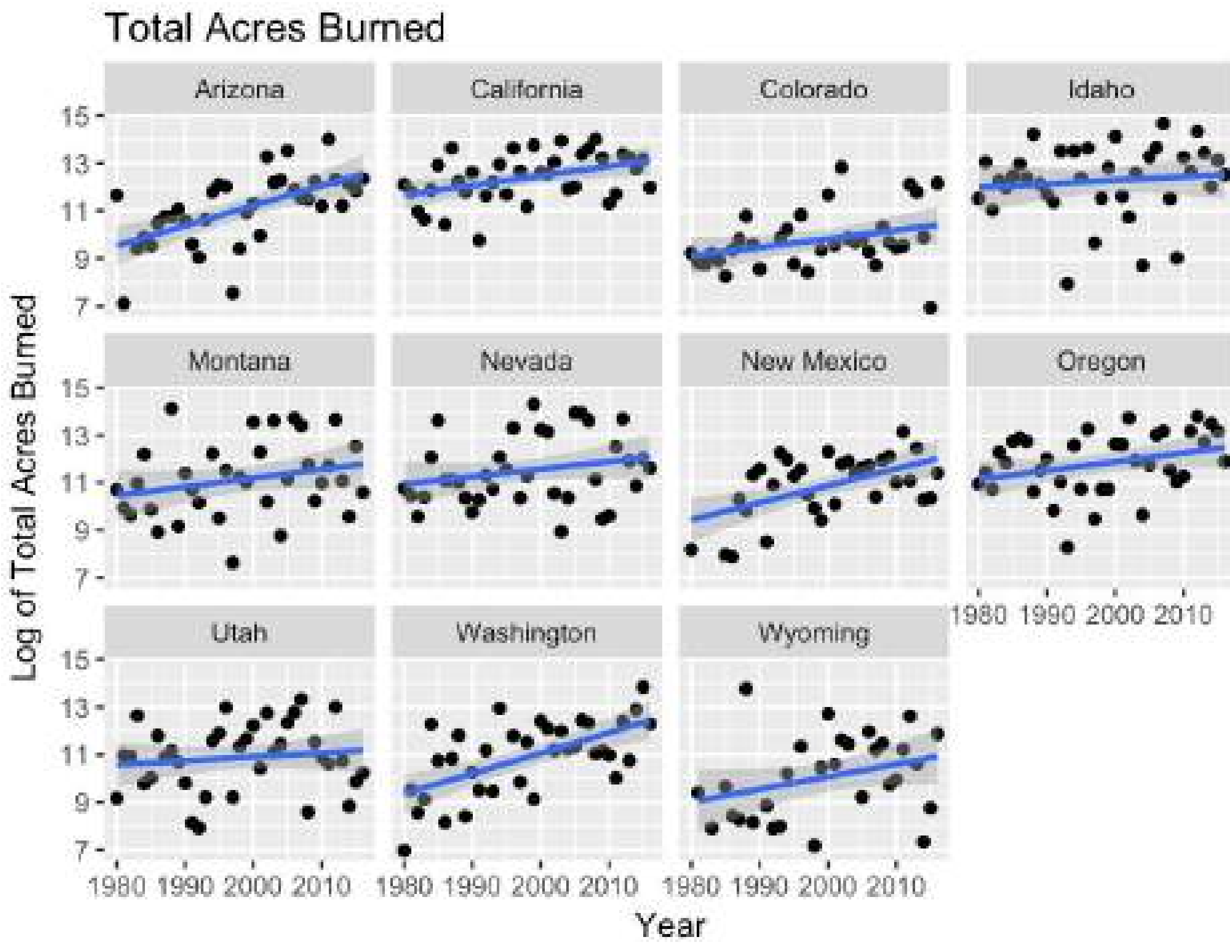
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(STATE, YR) %>%
summarize(totalacres = sum(ACRES)) %>%
**ggplot(mapping = aes(x=YR, y=log(totalacres))) + geom_point() +
facet_wrap(~STATE) + geom_smooth(method=lm, se=TRUE) +
ggtitle("Total Acres Burned") + xlab("Year") + ylab("Log of Total Acres
Burned")**

15. You can check your work against the solution file
CS1_Exercise2B.R

.

16. Save the script and then run it to see the output shown in the screenshot below. It's clear from this graph that there has been an increase in the acreage burned over the past few decades for all the states in the study area.



17. You may have wondered if there is a difference in the size of wildfires that were caused naturally as opposed to human induced. In the next few steps we'll write a script to do just that. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise2C.R.

18. At the top of the script, load the packages that will be used in this exercise.

library(readr) library(dplyr) library(ggplot2)

19. The first few lines of this script will be similar to the previous exercises, so I won't discuss the details of each line. Add the lines shown below.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE) df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
```

20. For this script we’ll filter so that only Natural and Human values are selected from the CAUSE column in addition to requiring that only fires greater than 1,000 acres be included.

There are additional values in the CAUSE column including UNKNOWN and a few other random values so that’s why we’re taking this extra step. The dataset does not include prescribed fires, so we don’t have to worry about that in this case.

The %in%
operator can be used with a vector in R to define multiple values as is the case here.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000 & CAUSE %in% c(‘Human’, ‘Natural’)) %>%
```

21. Group the data by CAUSE and YR.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000 & CAUSE %in% c(‘Human’, ‘Natural’)) %>%
group_by(CAUSE, YR) %>%
```

22. Sum the total acreage burned.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%
```

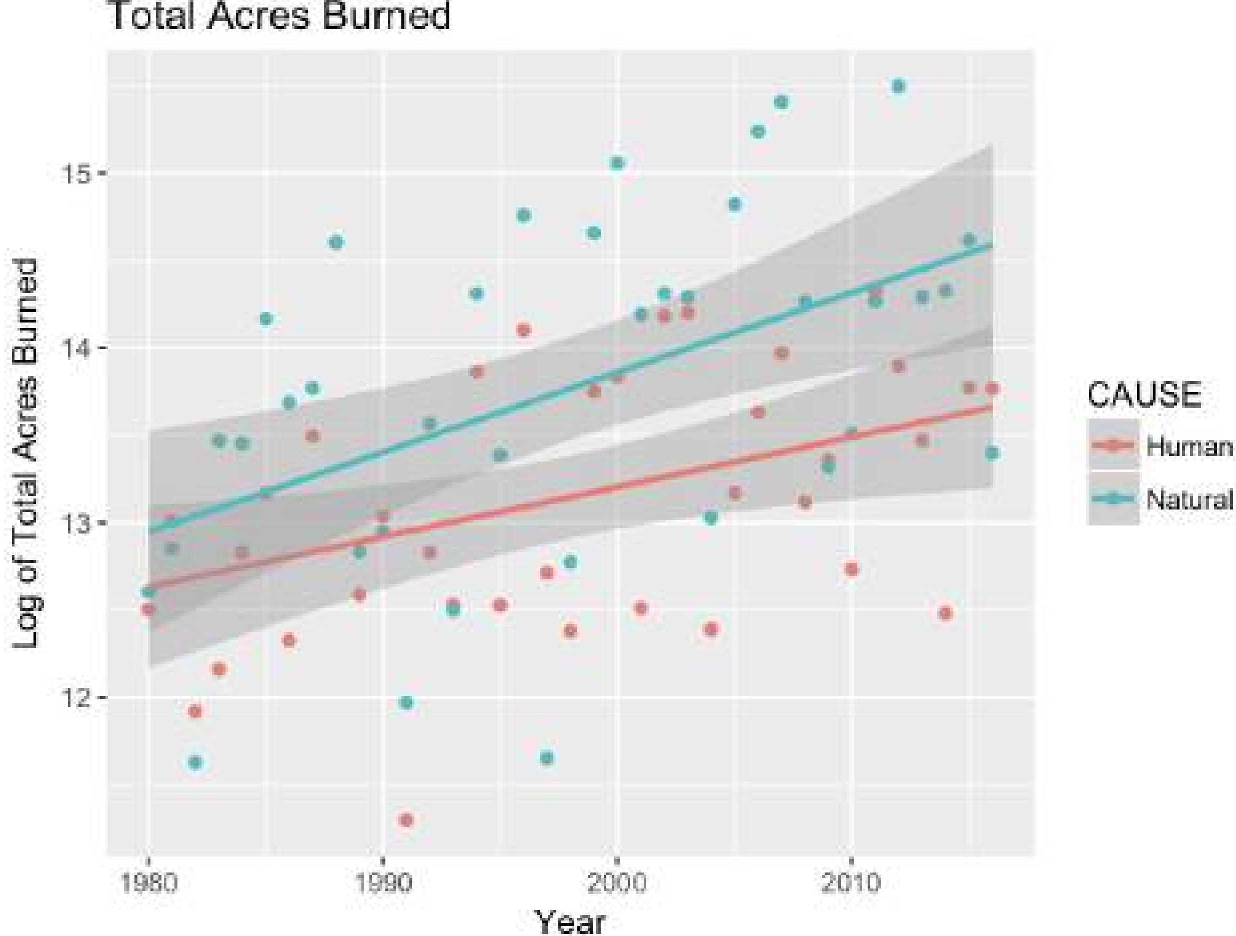
```
select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000 & CAUSE %in% c('Human', 'Natural')) %>%
group_by(CAUSE, YR) %>%
```
summarize(totalacres = sum(ACRES)) %>%

23. Plot the dataset. Use the
colour property from the aes() function to color code the values by CAUSE
.

```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE) %>%
filter(ACRES >= 1000 & CAUSE %in% c('Human', 'Natural')) %>%
group_by(CAUSE, YR) %>%
summarize(totalacres = sum(ACRES)) %>%
ggplot(mapping = aes(x=YR, y=log(totalacres), colour=CAUSE)) +
geom_point() + geom_smooth(method=lm, se=TRUE) + ggtitle("Total Acres
Burned") + xlab("Year") + ylab("Log of Total Acres Burned")
```

24. You can check your work against the solution file CS1_Exercise2C.R.
25. Save the script and then run it to see the output shown in the screenshot below. Both human and naturally caused wildfires have seen a significant increase in the amount of acreage burned over the past few decades, but the amount of acreage burned by naturally occurring fires appear to be increasing at a more rapid pace.



26. Finally, let’s create a violin plot to see the distribution of acres burned by state. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise2D.R.

27. At the top of the script, load the packages that will be used in this exercise.

```
library(readr)
library(dplyr)
library(ggplot2)
```

28. The first few lines of this script will be similar to the previous exercises, so I won’t discuss the details of each line. Add the lines shown below.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES,
```

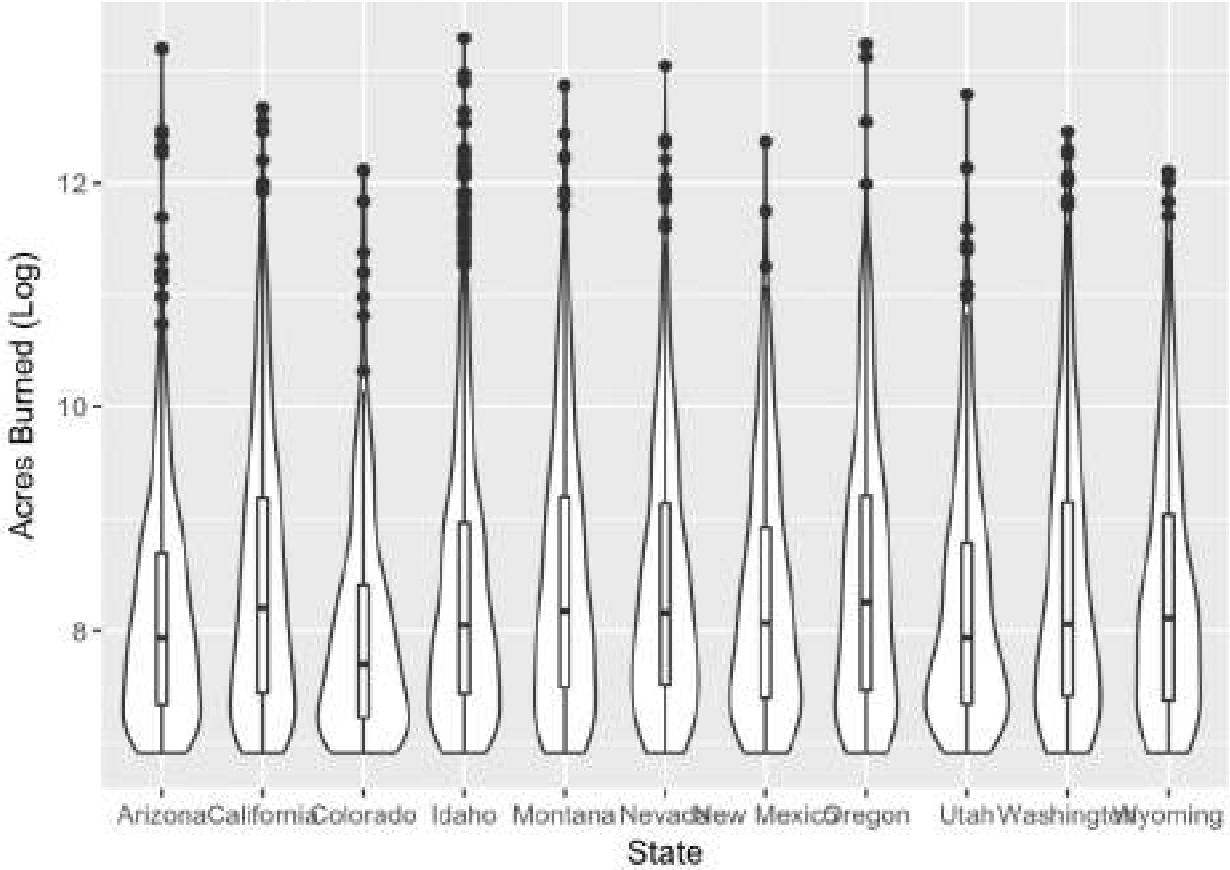
CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(STATE) %>%

29. Create a violin plot with an embedded box plot.

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES,
CAUSE) %>%
filter(ACRES >= 1000) %>%
group_by(STATE) %>%
ggplot(mapping = aes(x=STATE, y=log(ACRES))) + geom_violin() + geom_boxplot(width=0.1) + ggtitle("Wildfires by State Greater than 1,000 Acres") + xlab("State") + ylab("Acres Burned (Log)")

30. You can check your work against the solution file CS1_Exercise2D.R. 31. Save the script and then run it to see the output shown in the screenshot below.



Exercise 3: Is the size of individual wildfires increasing over time?

In the last exercise we found that the number of wildfires appears to be increasing over the past few decades. In this exercise we'll determine whether the size of those fires has increased as well. The StudyArea.csvfile contains a TOTALACRESc olumn that defines the number of acres burned by each fire. We'll group the fires by year and then by decade and determine the mean and median fire size for each.

1. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise3.R
. 2. At the top of the script, load the packages that will be used in this exercise.

library(readr) library(dplyr) library(ggplot2)

3. The first few lines of this script will be the same as the previous exercises, so I won't discuss the details of each line. By now you should be able to determine

what each of these lines will accomplish anyway. Add the lines shown below.

dfWildfires <- read_csv("StudyArea.csv", col_types = list(UNIT = col_character()), col_names = TRUE)
df <- select(dfWildfires, STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE)
df <- filter(df, ACRES >= 1000)
grp <- group_by(df, CAUSE, YR)

4. Summarize the data by determining the mean acreage burned for each group.
sm <- summarize(grp, mean(ACRES))

5. The summarize() function will create a new column called mean(ACRES) and add it to the output data frame. This isn't exactly a user-friendly name, so we'll change the name of this column in the next step. You can see the output of the summarize() function in the screenshot below.

	CAUSE	YR	mean(ACRES)
1	Human	1980	4896.345
2	Human	1981	5310.226
3	Human	1982	3572.143
4	Human	1983	3603.340
5	Human	1984	5926.952

6. Change the column name.
colnames(sm)[3] <- 'MEAN'
7. Create a scatterplot of the results.

ggplot(data=sm, mapping = aes(x=YR, y=MEAN)) + geom_point() + geom_smooth(method=lm, se=TRUE) + ggtitle("Average Size of Wildfires Has Increased for both Human and Natural Causes") + xlab("Year") + ylab("Average Wildfire Size")

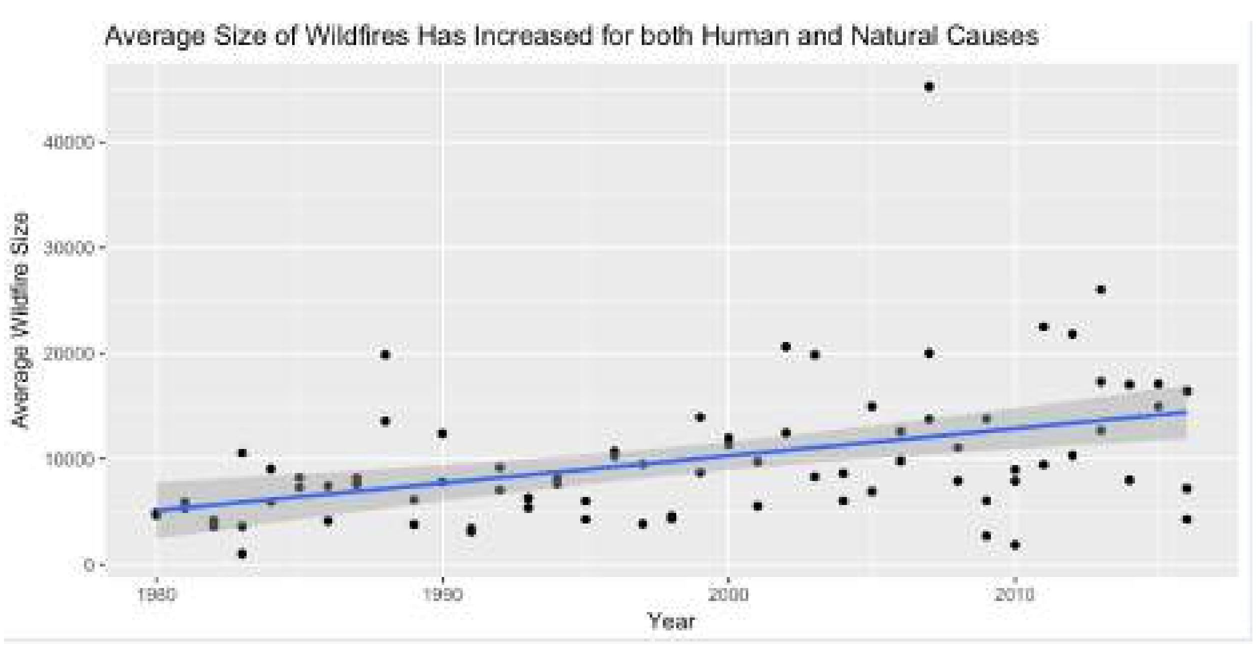
8. The entire script should appear as seen below.

```
library(readr)
library(dplyr)
library(ggplot2)

dfWildfires <- read_csv(“StudyArea.csv”, col_types = list(UNIT =
col_character()), col_names = TRUE)
df = select(dfWildfires, STATE, YR = YEAR_, ACRES = TOTALACRES,
CAUSE)
df <- filter(df, ACRES >= 1000)
grp <- group_by(df, CAUSE, YR)
sm <- summarize(grp, mean(ACRES))
colnames(sm)[3] <- ‘MEAN’
ggplot(data=sm, mapping = aes(x=YR, y=MEAN)) + geom_point() +
geom_smooth(method=lm, se=TRUE) + ggtitle(“Average Size of Wildfires Has
Increased for both Human and Natural Causes”) + xlab(“Year”) + ylab(“Average
Wildfire Size”)
```

9. You can check your work against the solution file
CS1_Exercise3.R
.

10. Save and run the script. If everything has been coded correctly you should see the following output. This graph indicates a clear trend toward larger wildfires over time.



11. Now let's look group the wildfires by decade, sum the total acreage burned during that time, and create a bar chart to display the results. 12. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise3B.R
. 13. At the top of the script, load the packages that will be used in this exercise.

```
library(readr)
library(dplyr)
library(ggplot2)
```

14. Load, select, and filter the data in the same way we've done with the other exercises in this chapter.

```
dfWildfires <- read_csv("StudyArea.csv", col_types = list(UNIT =
col_character()), col_names = TRUE)
df <- select(dfWildfires, ORGANIZATI, STATE, YR = YEAR_, ACRES =
TOTALACRES, CAUSE)
df <- filter(df, ACRES >= 1000)
```

15. In this step we'll use the mutate() function along with an ifelse() function to create a new column called DECADE and then populate the contents of this column based on the value of the YR column for each row. Add the code you see below.

```
df <- mutate(df, DECADE = ifelse(YR %in% 1980:1989, "1980-1989",
ifelse(YR %in% 1990:1999, "1990-1999", ifelse(YR %in% 2000:2009, "2000-
2009", ifelse(YR %in% 2010:2016, "2010-2016", "-99")))))
```

16. Group the dataset by
DECADE.

```
grp <- group_by(df, DECADE)
```

17. Summarize the data by calculating the mean value of acres burned.

```
sm <- summarize(grp, mean(ACRES))
```

18. Rename the column created by the summarize() function.z

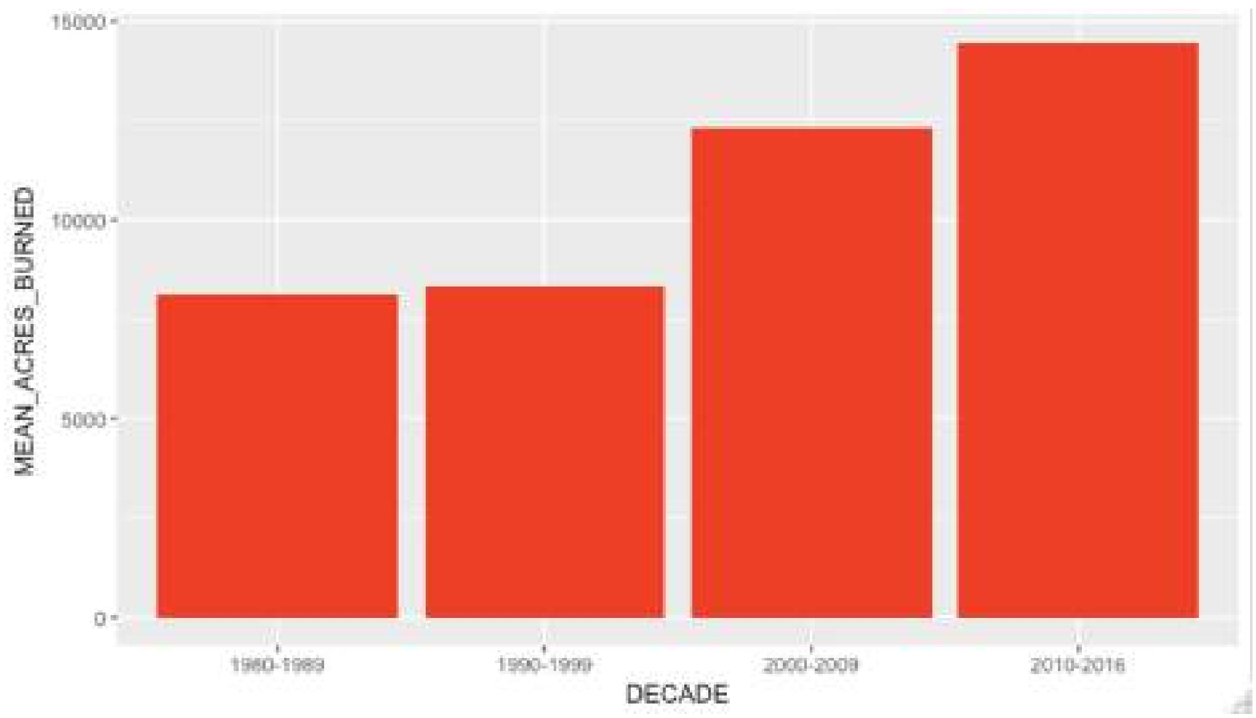
```
names(sm) <- c("DECADE", "MEAN_ACRES_BURNED")
```

19. Use the geom_col() function along with ggplot() to create a bar chart that displays the mean wildfire size by decade.

```
ggplot(data=sm) + geom_col(mapping = aes(x=DECADE, y=MEAN_ACRES_
BURNED), fill="red")
```

20. You can check your work against the solution file CS1_Exercise3B.R
.

21. Save and run the script. If everything has been coded correctly you should see the following output. This bar chart indicates a clear trend toward larger wildfires with each passing decade, although it should be noted that the dataset only extends through 2016 so the results for the current decade may be different in a few years.



Exercise 4: Has the length of the fire season increased over time?

Wildfire season is generally defined as the time period between the year's first and last large wildfires. The infographic below, from the Union of Concerned Scientists (https:// www.ucsusa.org/global-warming/science-and-impacts/impacts/infographic-wildfiresclimate-change.html#.W1cji9hKj_Q), highlights the length of the wildfire season for the Western U.S. as a region. Local wildfire seasons vary by location but have almost universally become longer over the past 40 years.

In this exercise we'll measure the length of the wildfire season over the past few decades for the region as a whole, as well as individual states.
1. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise4.R.

2. At the top of the script, load the packages that will be used in this exercise. Note that you will need to load the lubridate library for this exercise since we'll be dealing with dates.

```
library(readr)
library(dplyr)
library(lubridate)
library(ggplot2)
```

3. The first few lines of this script will be similar to the previous exercises, so I won't discuss the details of each line. By now you should be able to determine what each of these lines will accomplish anyway. Add the lines shown below to load the data, select the columns, and filter the data.

```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES,
CAUSE, STARTDATED) %>%
filter(ACRES >= 1000) %>%
```

4. To measure the length of the wildfire season we're going to convert the start date of each fire into the day of the year. For example, if a fire occurred on

February 1st, it would be the 32nd day of the year. Use the mutate() function as seen below to accomplish this. The mutate() function uses the yday() lubridate function to convert the value for the STARTDATED column into the day of the year.

```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES,
CAUSE, STARTDATED) %>%
filter(ACRES >= 1000) %>%
mutate(DOY = yday(as.Date(STARTDATED, format='%m/%d/%y
%H:%M')))

%>%
```

5. Group the data by year.

```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES,
CAUSE, STARTDATED) %>%
filter(ACRES >= 1000) %>%
mutate(DOY = yday(as.Date(STARTDATED, format='%m/%d/%y %H:%M')))

%>%
group_by(YR) %>%
```

6. Get the earliest and latest start dates of the wildfires using the summarize() function.

```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES,
CAUSE, STARTDATED) %>%
filter(ACRES >= 1000) %>%
```

mutate(DOY = yday(as.Date(STARTDATED, format='%m/%d/%y %H:%M')))

%>%
group_by(YR) %>%
summarize(dtEarly = min(DOY, na.rm=TRUE), dtLate = max(DOY,

na.rm=TRUE)) %>%

7. Finally, use ggplot with two calls to geom_line() to create two line graphs that display the earliest start and latest end dates by year. You'll also add a smoothed regression line to both line graphs.

df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES,
CAUSE, STARTDATED) %>%
filter(ACRES >= 1000) %>%
mutate(DOY = yday(as.Date(STARTDATED, format='%m/%d/%y %H:%M')))

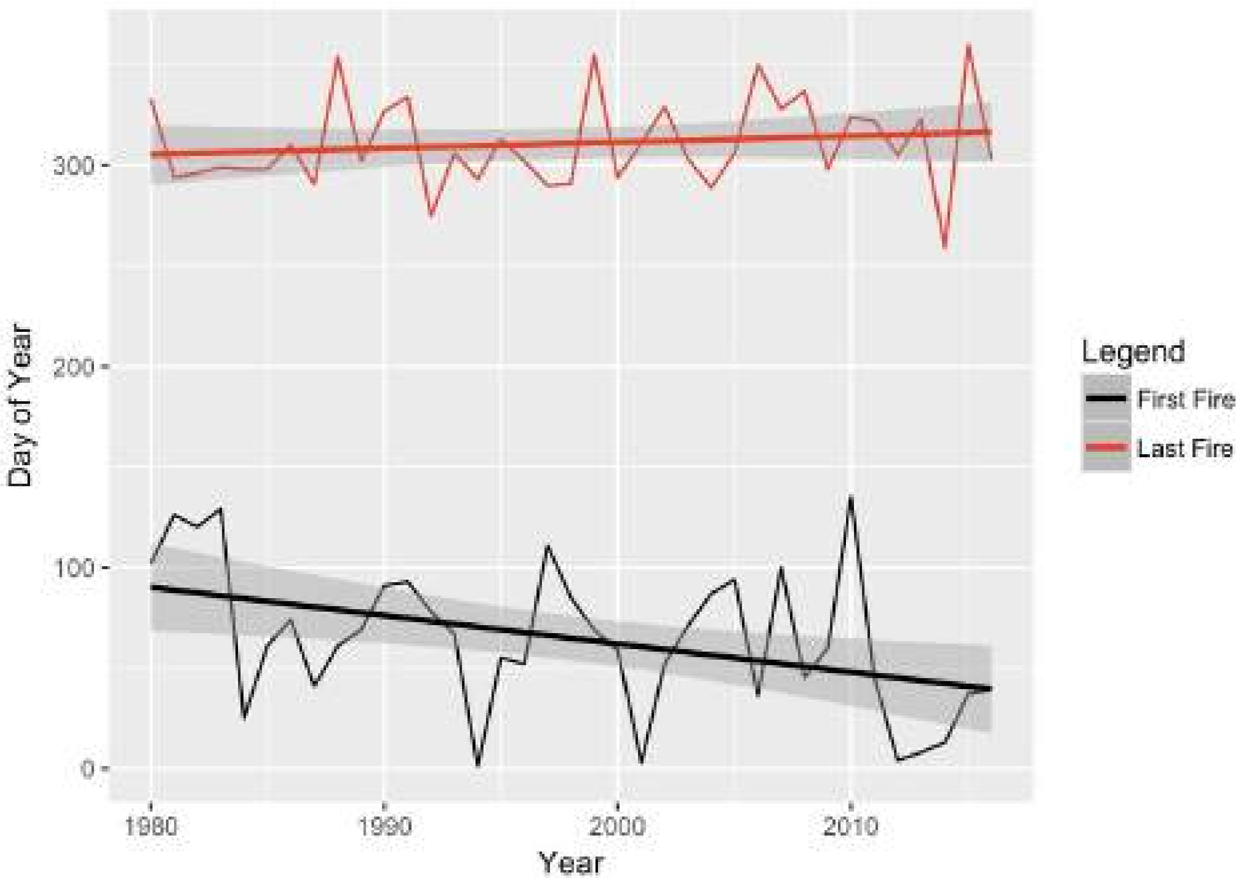
%>%
group_by(YR) %>%
summarize(dtEarly = min(DOY, na.rm=TRUE), dtLate = max(DOY,

na.rm=TRUE)) %>%
**ggplot() + geom_line(mapping = aes(x=YR, y=dtEarly, color='B')) +
geom_line(mapping = aes(x=YR, y=dtLate, color='R')) + geom_
smooth(method=lm, se=TRUE, aes(x=YR, y=dtEarly, color="B")) +
geom_smooth(method=lm, se=TRUE, aes(x=YR, y=dtLate, color="R")) +
xlab("Year") + ylab("Day of Year") + scale_colour_manual(name =
"Legend", values = c("R" = "#FF0000", "B" = "#000000"), labels = c("First
Fire", "Last Fire"))**

8. You can check your work against the solution file
CS1_Exercise4.R
.

9. Save and run the script. If everything has been coded correctly you should see the following output. This chart shows a clear lengthening of the wildfire season

with the first fire date coming significantly earlier in recent years and the start date of the last fire increasing as well.



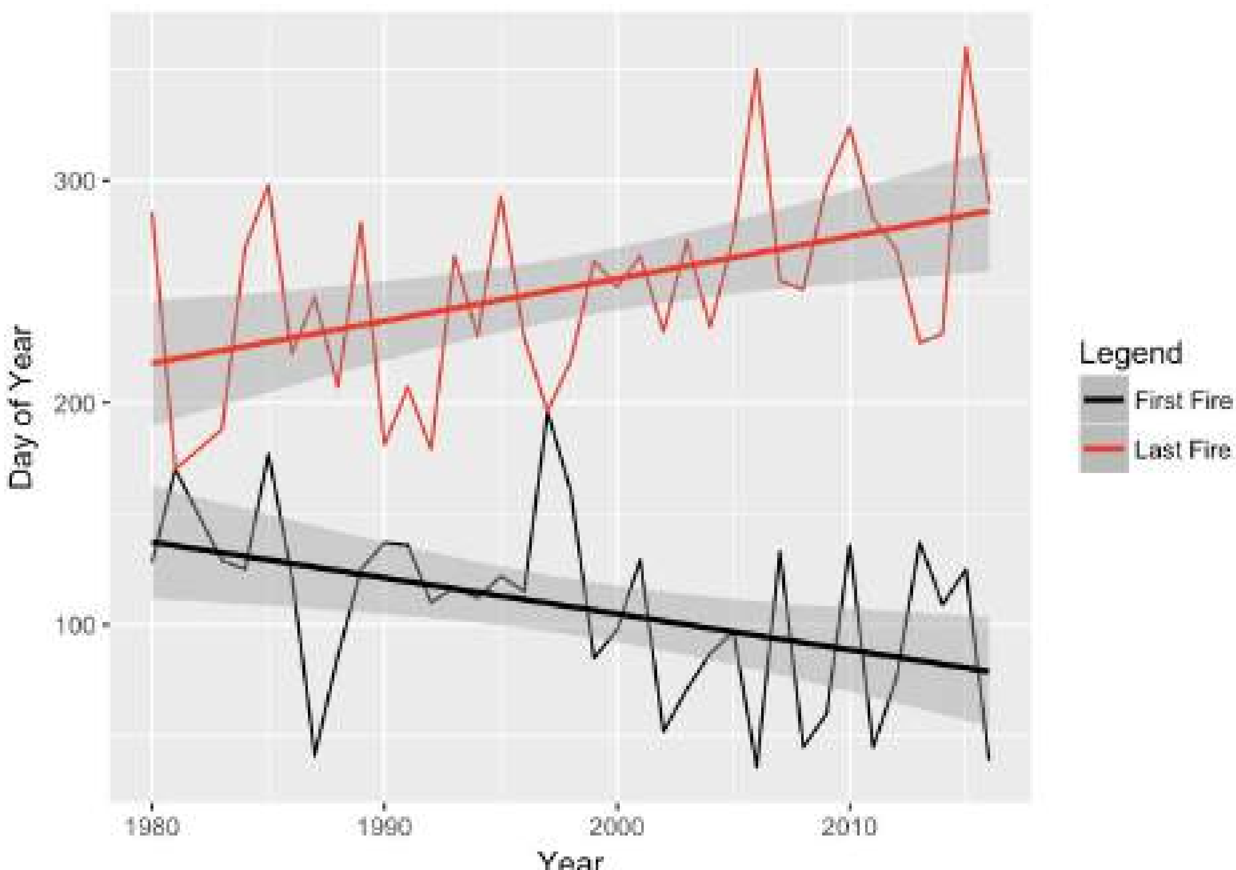
10. The last script examined the trends in wildfire season length for the entire study area, but you might want to examine these trends at a state level instead. This can be easily accomplished by adding a second statement to the filter. Update the filter as seen below and re-run the script to see the result.

df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%
select(ORGANIZATI, STATE, YR = YEAR_, ACRES = TOTALACRES, CAUSE, STARTDATED) %>%
filter(ACRES >= 1000 & STATE == ‘Arizona’) %>%
mutate(DOY = yday(as.Date(STARTDATED, format=’%m/%d/%y %H:%M’)))

%>%
group_by(YR) %>%

summarize(dtEarly = min(DOY, na.rm=TRUE), dtLate = max(DOY,

na.rm=TRUE)) %>%
ggplot() + geom_line(mapping = aes(x=YR, y=dtEarly, color='B')) + geom_line(mapping = aes(x=YR, y=dtLate, color='R')) + geom_smooth(method=lm, se=TRUE, aes(x=YR, y=dtEarly, color="B")) + geom_smooth(method=lm, se=TRUE, aes(x=YR, y=dtLate, color="R")) + xlab("Year") + ylab("Day of Year") + scale_colour_manual(name = "Legend", values = c("R" = "#FF0000", "B" = "#000000"), labels = c("First Fire", "Last Fire"))



The State of Arizona shows an even bigger trend toward longer wildfire seasons. Try a few other states as well.

Exercise 5: Does the average wildfire size differ by federal organization

To wrap up this chapter we'll examine if the average wildfire size differs by federal organization. The StudyArea.csvfile includes a column (ORGANIZATI)

that indicates the jurisdiction where the fire started. This column can be used to group the wildfires.

1. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise5.R
. 2. At the top of the script, load the packages that will be used in this exercise.

```
library(readr) library(dplyr) library(ggplot2)
```

3. The first few lines of this script will be similar to the previous exercises, so I won’t discuss the details of each line. By now you should be able to determine what each of these lines will accomplish anyway. Add the lines shown below to load the data, select the columns, and filter the data.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORG = ORGANIZATI, STATE, YR = YEAR_, ACRES =
TOTALACRES, CAUSE, STARTDATED) %>%
filter(ACRES >= 1000) %>%
```

4. Group the dataset by
ORG and YR
.

```
df <- read_csv(“StudyArea.csv”, col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORG = ORGANIZATI, STATE, YR = YEAR_, ACRES =
TOTALACRES,

CAUSE, STARTDATED) %>%
filter(ACRES >= 1000 & ORG %in% c(‘BIA’, ‘BLM’, ‘FS’, ‘FWS’, ‘NPS’))
%>%
group_by(ORG, YR) %>%
```

5. Summarize the data by calculating the mean acreage burned by organization and year.

```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

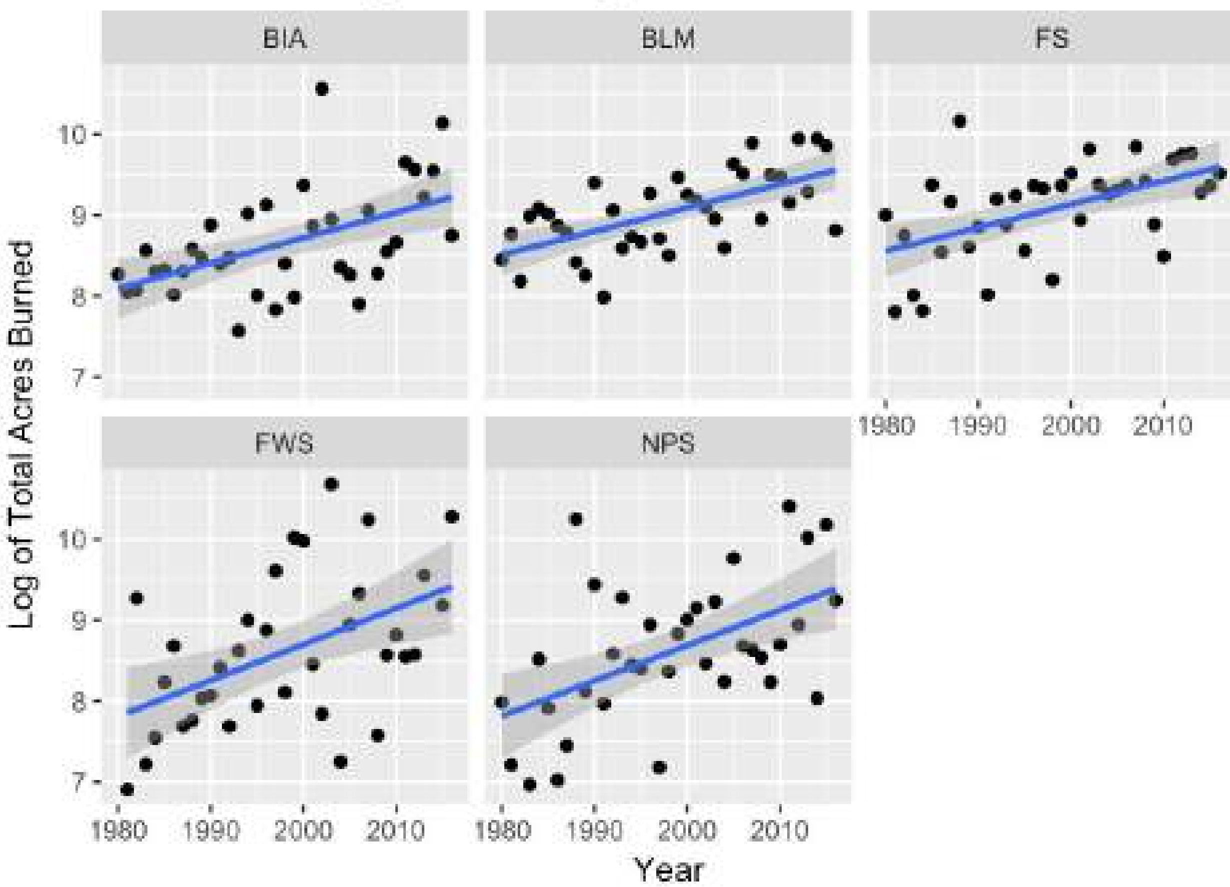
select(ORG = ORGANIZATI, STATE, YR = YEAR_, ACRES =
TOTALACRES, CAUSE, STARTDATED) %>%
filter(ACRES >= 1000 & ORG %in% c('BIA', 'BLM', 'FS', 'FWS', 'NPS'))
%>%
group_by(ORG, YR) %>%
summarize(meanacres = mean(ACRES)) %>%
```

6. Create a facet plot for the mean acreage burned by year for each organization.

```
df <- read_csv("StudyArea.csv", col_types =
list(UNIT = col_character()), col_names = TRUE)
df %>%

select(ORG = ORGANIZATI, STATE, YR = YEAR_, ACRES =
TOTALACRES, CAUSE, STARTDATED) %>%
filter(ACRES >= 1000 & ORG %in% c('BIA', 'BLM', 'FS', 'FWS', 'NPS'))
%>%
group_by(ORG, YR) %>%
summarize(meanacres = mean(ACRES)) %>%
ggplot(mapping = aes(x=YR, y=log(meanacres))) + geom_ point()+
facet_wrap(~ORG) + geom_smooth(method=lm, se=TRUE) + ggtitle("Acres
Burned by Federal Organization") + xlab("Year") + ylab("Log of Total
Acres Burned")
```

7. You can check your work against the solution file CS1_Exercise5.R.
8. Save and run the script. If everything has been coded correctly you should see the following output. It appears as though all the federal agencies have experienced similar increases in the size of wildfires since 1980.



Chapter 11

Case Study – Single Family Residential Home and Rental Values

The Zillow Research group publishes several different measures of homes values on a monthly basis including median list prices, median sale prices, and the Zillow Home Value Index (ZHVI). The ZHVI is based on Zillow's internal methodology for measuring home values over time. In addition, Zillow also publishes a similar measure of rental values (ZRI) as well as a number of other real estate related datasets.

The methodology for ZHVI can be read in detail at https://www.zillow.com/research/zhvi-methodology-6032/, but the simple explanation is that Zillow takes all estimated home values for a given region and month (Zestimate), takes a median of these values, applies some adjustments to account for seasonality or errors in individual home estimates, and then does the same across all months over the past 20 years and for many different geography levels (ZIP, neighborhood, city, county, metro, state, and country). For example, if ZHVI was $400,000 in Seattle one month, that indicates that 50 percent of homes in the area are worth more than $400,000 and 50 percent are worth less (adjusting for seasonal fluctuations– e.g. prices tend to be low in December).

Zillow recommends using ZHVI to track home values over time for the very simple reason that ZHVI represents the whole housing stock and not just the homes that list or sell in a given month. Imagine a month where no homes outside of California sold. A national median price series or median list series would both spike. ZHVI, however, would remain a median of all homes across the country and wouldn't skew toward California any more than in the previous month. ZHVI will always reflect the value of all homes and not just the ones that list or sell in a given month. In this chapter we'll use some basic R visualization techniques to better understand residential real estate values and rental prices in the Austin, TX metropolitan area.

In this chapter we'll cover the following topics:

- What is the trend for home values in the Austin metropolitan area?
- What is the trend for rental values in the Austin metropolitan area?
- Determining the price-rent ratio for the Austin metropolitan area.

• Comparing residential home values in Austin to other Texas metropolitan areas

Exercise 1: What is the trend for home values in the Austin metro area

The County_Zhvi_SingleFamilyResidence.csv file in your IntroR\Data folder contains home value data from Zillow. The Zillow Home Value Index (ZHVI) is a smoothed, seasonally adjusted measure of the median estimated home value across a given region and housing type. It is a dollar-denominated alternative to repeat-sales indices. Zillow also publishes home value and other housing data for local markets, as well as a more detailed methodology and a comparison of ZHVI to the S&P CoreLogic Case-Shiller Home Price Indices. We'll use this file for this particular exercise.

In this first exercise we'll examine home values over the past couple of decades from the Austin metropolitan area.

1. In your IntroR folder create a new folder called CaseStudy2. You can do this inside RStudio by going to the **Files** pane and selecting **New Folder** inside your working directory.

2. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS2_Exercise1.R.
3. At the top of the script, load the packages that will be used in this exercise.

```
library(readr) library(dplyr) library(ggplot2)
```

4. Use the
read_csv() function from the readr
package to load the data into a data frame.
```
df <- read_csv("County_Zhvi_SingleFamilyResidence.csv",col_names = TRUE)
```
5. Start a piping expression and define the columns that should be included in the data frame.
```
df %>%

select(RegionName, State, Metro, `1996` = `1996-05`, `1997` = `1997-05`,
`1998` = `1998-05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-
05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-05`, `1999` =
`1999-05`,`2000` = 05`, `2007` = `2007-05`, `2008` = `2008-05`, `2009` =
```

`2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012` = `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`, `2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>%

6. Filter the data frame to include the Austin metropolitan area from the state of Texas.

df %>%
select(RegionName, State, Metro, `1996` = `1996-05`, `1997` = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = 05`, `2007` = `2007-05`, `2008` = `2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012` = `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`, `2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>% **filter(State == 'TX' & Metro == 'Austin') %>%**

7. If you were to view the structure of the data frame at this point it would look like the screenshot below. A common problem in many datasets is that the column names are not variables but rather values of a variable. In the figure provided below, the columns that represent each year in the study are actually values of the variable YEAR. Each row in the existing table actually represents many annual observations. The tidyr package can be used to gather these existing columns into a new variable. In this case, we need to create a new column called YR and then gather the existing values in the annual columns into the new YR column.

In the next step we'll use the gather() function to accomplish this.

	RegionName	State	Metro	1996	1997	1998	1999	2000	2001	2002	2003
1	Travis	TX	Austin	165500	169700	168300	163700	184100	189400	186100	186500
2	Williamson	TX	Austin	157000	157900	163000	159500	164200	171100	174500	175000
3	Hays	TX	Austin	134900	133300	137800	142000	149800	150700	152000	155500
4	Bastrop	TX	Austin	70200	76500	77900	82700	90300	92500	100400	106100
5	Caldwell	TX	Austin	46700	53100	59300	71500	73100	74800	80500	90700

8. Use the gather() function to tidy up the data so that a new YR column is created, and rows for each county (RegionName) and year value are added.

df <- read_csv("County_Zhvi_SingleFamilyResidence.csv",col_names = TRUE)
df %>%

select(RegionName, State, Metro, `1996` = `1996-05`, `1997` = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = 05`, `2007` = `2007-05`, `2008` = `2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012` = `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`, `2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>% filter(State == ‘TX’ & Metro == ‘Austin’) %>%
gather(`1996`, `1997`, `1998`, `1999`, `2000`, `2001`, `2002`, `2003`, `2004`, `2005`, `2006`, `2007`, `2008`, `2009`, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`, `2018`,key=’YR’, value=’ZHVI’) %>%

9. If you were to view the result, the data frame would now appear as seen in the figure below.

	RegionName	State	Metro	YR	ZHVI
1	Travis	TX	Austin	1996	165500
2	Williamson	TX	Austin	1996	157000
3	Hays	TX	Austin	1996	134900
4	Bastrop	TX	Austin	1996	70200
5	Caldwell	TX	Austin	1996	46700
6	Travis	TX	Austin	1997	169700
7	Williamson	TX	Austin	1997	157900
8	Hays	TX	Austin	1997	133300
9	Bastrop	TX	Austin	1997	76500
10	Caldwell	TX	Austin	1997	53100
11	Travis	TX	Austin	1998	168300
12	Williamson	TX	Austin	1998	163000
13	Hays	TX	Austin	1998	137800
14	Bastrop	TX	Austin	1998	77900
15	Caldwell	TX	Austin	1998	59300
16	Travis	TX	Austin	1999	163700
17	Williamson	TX	Austin	1999	159500
18	Hays	TX	Austin	1999	142000
19	Bastrop	TX	Austin	1999	82700
20	Caldwell	TX	Austin	1999	71500

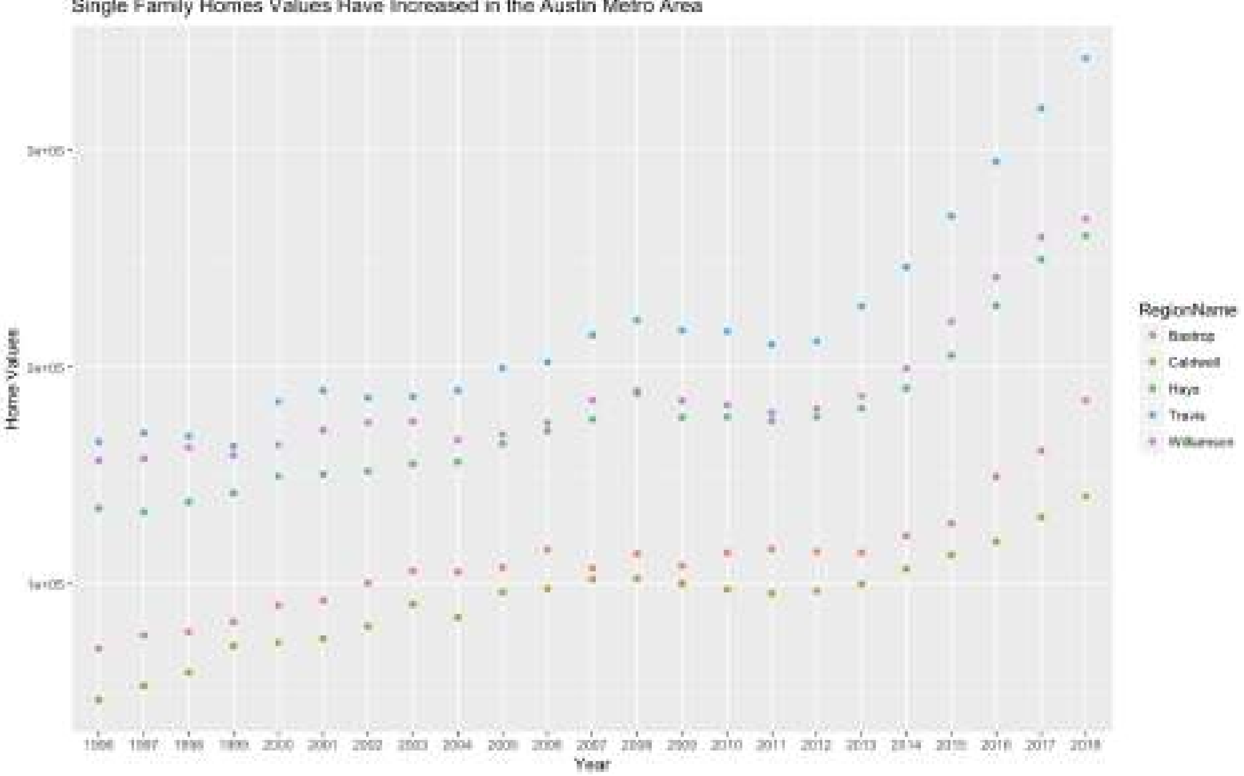
10. Now we’re ready to plot the data. Add the code you see below to create a point plot that is grouped by RegionName
(County).

df <- read_csv("County_Zhvi_SingleFamilyResidence.csv",col_names = TRUE)
df %>%

select(RegionName, State, Metro, `1996` = `1996-05`, `1997` = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = = `1997-05`, `1998` = `1998-05`, `1999` = `1999-05`,`2000` = 05`, `2007` = `2007-05`, `2008` = `2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012` = `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`, `2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>% filter(State == 'TX' & Metro == 'Austin') %>%
gather(`1996`, `1997`, `1998`, `1999`, `2000`, `2001`, `2002`, `2003`, `2004`, `2005`, `2006`, `2007`, `2008`, `2009`, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`, `2018`,key='YR', value='ZHVI') %>%
ggplot(mapping = aes(x=YR, y=ZHVI, colour=RegionName)) + geom_ point() + geom_smooth(method=lm, se=TRUE) + ggtitle("Single Family Homes Values Have Increased in the Austin Metro Area") + xlab("Year") + ylab("Home Values")

11. You can check your work against the solution file
CS2_Exercise1.R
.

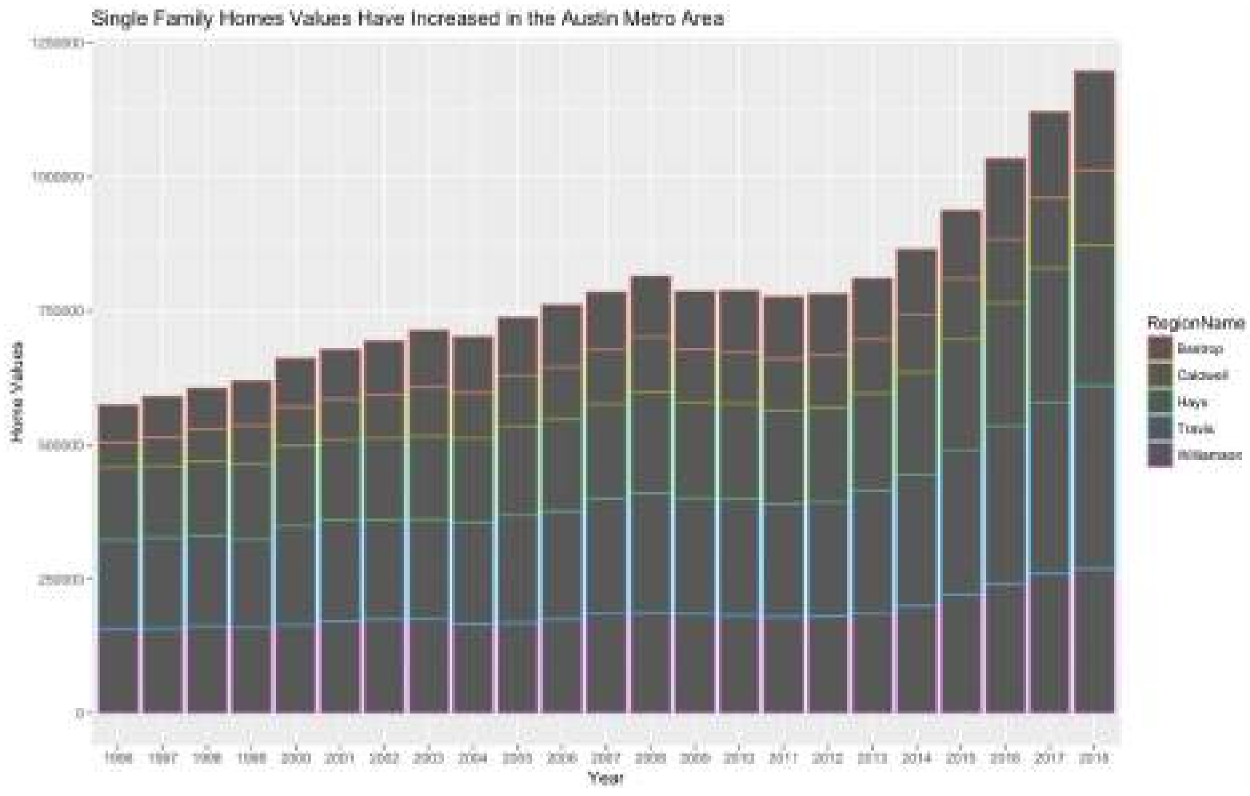
12. Save the script and then run it to see the output shown in the screenshot below. All counties in the Austin metropolitan area have experienced significantly increased values in the past couple decades. The increase has been particularly noticeable since 2012.



13. Instead of a simple dot plot you might want to create a bar chart instead. Comment out the line of code that calls the existing ggplot() function and add a new line as seen below.

ggplot(mapping = aes(x=YR, y=ZHVI, colour=RegionName)) + geom_ col() + ggtitle("Single Family Homes Values Have Increased in the Austin Metro Area") + xlab("Year") + ylab("Home Values")

14. Save and run the script and the output should now appear as seen in the screenshot below. The upward trend in values seems even more obvious when viewed in this manner.



Exercise 2: What is the trend for rental rates in the Austin metro area?

The County_Zri_SingleFamilyResidenceRental.csvfile in your IntroR\Data folder contains single family residential real estate values Zillow. Zillow Rent Index (ZRI) is a smoothed, seasonally adjusted measure of the median estimated market rate rent across a given region and housing type. ZRI is a dollar-denominated alternative to repeatrent indices.

In this exercise we’ll examine rent values over the past few years from the Austin metropolitan area.
1. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS2_Exercise2.R
. 2. At the top of the script, load the packages that will be used in this exercise.

library(readr) library(dplyr) library(ggplot2)

3. Use the
read_csv() function from the readr
package to load the data into a data frame.

df <- read_csv("County_Zhvi_SingleFamilyResidence.csv",col_names = TRUE)

4. Select the columns and filter the data. This dataset contains data from 2010 going forward. We'll use data from December of the years 2010 to 2017 for the Austin, TX metropolitan area.

df <- read_csv("County_Zri_SingleFamilyResidenceRental.csv",col_ names = TRUE)
df %>%

select(RegionName, State, Metro, `2010` = `2010-12`, `2011` = `2011-12`, `2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
filter(State == 'TX' & Metro == 'Austin') %>%

5. Gather the data.

df <- read_csv("County_Zri_SingleFamilyResidenceRental.csv",col_ names = TRUE)
df %>%

select(RegionName, State, Metro, `2010` = `2010-12`, `2011` = `2011-12`, `2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
filter(State == 'TX' & Metro == 'Austin') %>%
gather(`2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`,key='YR', value='ZRI') %>%

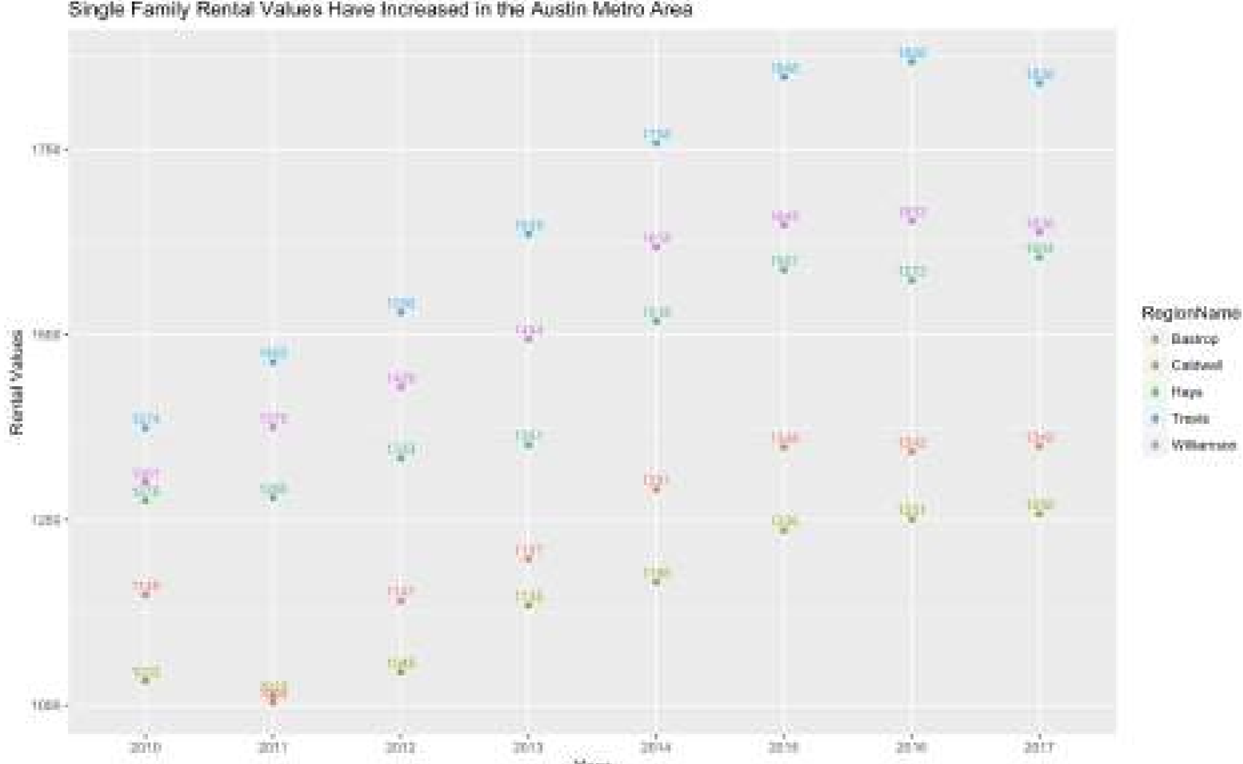
6. Call the
ggplot()
function to plot the data. In this plot we'll also add labels to each point.

df <- read_csv("County_Zri_SingleFamilyResidenceRental.csv",col_ names = TRUE)
df %>%

select(RegionName, State, Metro, `2010` = `2010-12`, `2011` = `2011-12`, `2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
filter(State == 'TX' & Metro == 'Austin') %>%

gather(`2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`,key='YR', value='ZRI') %>%
ggplot(mapping = aes(x=YR, y=ZRI, colour=RegionName)) + geom_ point() + geom_text(aes(label=ZRI, vjust = -0.5), size=3) + ggtitle("Single Family Rental Values Have Increased in the Austin Metro Area") + xlab("Year") + ylab("Rental Values")

7. You can check your work against the solution file CS2_Exercise2.R.
8. Save the script and then run it to see the output shown in the screenshot below.



Exercise 3: Determining the Price-Rent Ratio for the Austin metropolitan area

The price-to-rent ratio is a measure of the relative affordability of renting and buying in a given housing market. It is calculated as the ratio of home prices to annual rental rates. So, for example, in a real estate market where, on average, a home worth $200,000 could rent for $1000 a month, the price-rent ratio is 16.67. That's determined using the formula: $200,000 ÷ (12 x $1,000). In general, the lower the ratio, the more favorable to real estate investors looking for residential property.

In this exercise you'll join the Zillow home value data to the rental data, create a new column to hold the price-rent ratio, calculate the ratio, and plot the data as a bar chart. 1. In RStudio select
File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS1_Exercise3.R
. 2. At the top of the script, load the packages that will be used in this exercise.

```
library(readr) library(dplyr) library(ggplot2)
```

3. In this step you'll read the residential valuation information from the Zillow file, define the columns that should be used, filter the data and gather the data. In this case we're going to filter the data so that only Travis County is included. Add the following lines of code to your script to accomplish this task.

```
dfHomeVals <- read_csv("County_Zhvi_SingleFamilyResidence.
csv",col_names = TRUE)
dfHomeVals <- select(dfHomeVals, RegionName, State, Metro,`2010` =
dfHomeVals <- select(dfHomeVals, RegionName, State, Metro,`2010` = 12`,
`2014` = `2014-12`, `2015` = `2015-12`, `2016` = `2016-12`, `2017` = `2017-
12`)
dfHomeVals <- filter(dfHomeVals, State == 'TX' & Metro == 'Austin' &
RegionName == 'Travis')
dfHomeVals <- gather(dfHomeVals, `2010`, `2011`, `2012`, `2013`, `2014`,
`2015`, `2016`, `2017`,key='YR', value='ZHVI')
```

4. Now do the same for the rental data.

```
dfRentVals <- read_csv("County_Zri_SingleFamilyResidenceRental.
csv",col_names = TRUE)
dfRentVals <- select(dfRentVals, RegionName, State, Metro, `2010` = `2010-
12`, `2011` = `2011-12`, `2012` = `2012-12`, `2013` = = `2010-12`, `2011` =
`2011-12`, `2012` = `2012-12`, `2013` = 12`, `2017` = `2017-12`)
dfRentVals <- filter(dfRentVals, State == 'TX' & Metro == 'Austin' &
RegionName == 'Travis')
dfRentVals <- gather(dfRentVals, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`,
`2016`, `2017`,key='YR', value='ZRI')
```

5. The two previous steps created data frames for the residential home value and rental data. In this step we'll join those two data frames together using the dplyr package. Add the line of code you see below to your script. This uses the inner_join() function, which is the simplest type of join. An inner join matches pairs of observations whenever their keys are equal.

```
df <- inner_join(dfHomeVals, dfRentVals, by = 'YR')
```

6. If you were to view the resulting data frame at this point it would look like the screenshot below. Notice that the ZHVI (residential home value) and ZRI (rental value) columns are attached.

	RegionName.x	State.x	Metro.x	YR	ZHVI	RegionName.y	State.y	Metro.y	ZRI
1	Travis	TX	Austin	2010	213300	Travis	TX	Austin	1374
2	Travis	TX	Austin	2011	209500	Travis	TX	Austin	1463
3	Travis	TX	Austin	2012	220700	Travis	TX	Austin	1530
4	Travis	TX	Austin	2013	240500	Travis	TX	Austin	1635
5	Travis	TX	Austin	2014	260200	Travis	TX	Austin	1758
6	Travis	TX	Austin	2015	283200	Travis	TX	Austin	1848
7	Travis	TX	Austin	2016	312400	Travis	TX	Austin	1868
8	Travis	TX	Austin	2017	327400	Travis	TX	Austin	1839

7. Next, use the mutate() function to create a column called PriceRentRatio, and populate the rows using the calculation seen below.

```
df <- mutate(df, PriceRentRatio = ZHVI / (12 * ZRI))
```

8. If you were to view the results of the mutate() function it would appear as seen in the screenshot below. Notice that each year includes a PriceRentRatio value that has been calculated.

	RegionName.x	State.x	Metro.x	YR	ZHVI	RegionName.y	State.y	Metro.y	ZRI	PriceRentRatio
1	Travis	TX	Austin	2010	213300	Travis	TX	Austin	1374	12.93668
2	Travis	TX	Austin	2011	209500	Travis	TX	Austin	1463	11.93324
3	Travis	TX	Austin	2012	220700	Travis	TX	Austin	1530	12.02070
4	Travis	TX	Austin	2013	240500	Travis	TX	Austin	1635	12.25790
5	Travis	TX	Austin	2014	260200	Travis	TX	Austin	1758	12.33409
6	Travis	TX	Austin	2015	283200	Travis	TX	Austin	1848	12.77056
7	Travis	TX	Austin	2016	312400	Travis	TX	Austin	1868	13.93647
8	Travis	TX	Austin	2017	327400	Travis	TX	Austin	1839	14.83596

9. Finally, create a bar chart using geom_col() with PriceRentRatio as the y axis, and YR as the x
axis.

```
ggplot(data=df) + geom_col(mapping = aes(x=YR, y=PriceRentRatio),
fill=”red”)
```

10. Your entire script should appear as seen below.

```
library(readr)
library(dplyr)
library(ggplot2)

dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
dfHomeVals <- select(dfHomeVals, RegionName, State, Metro,`2010` =
```

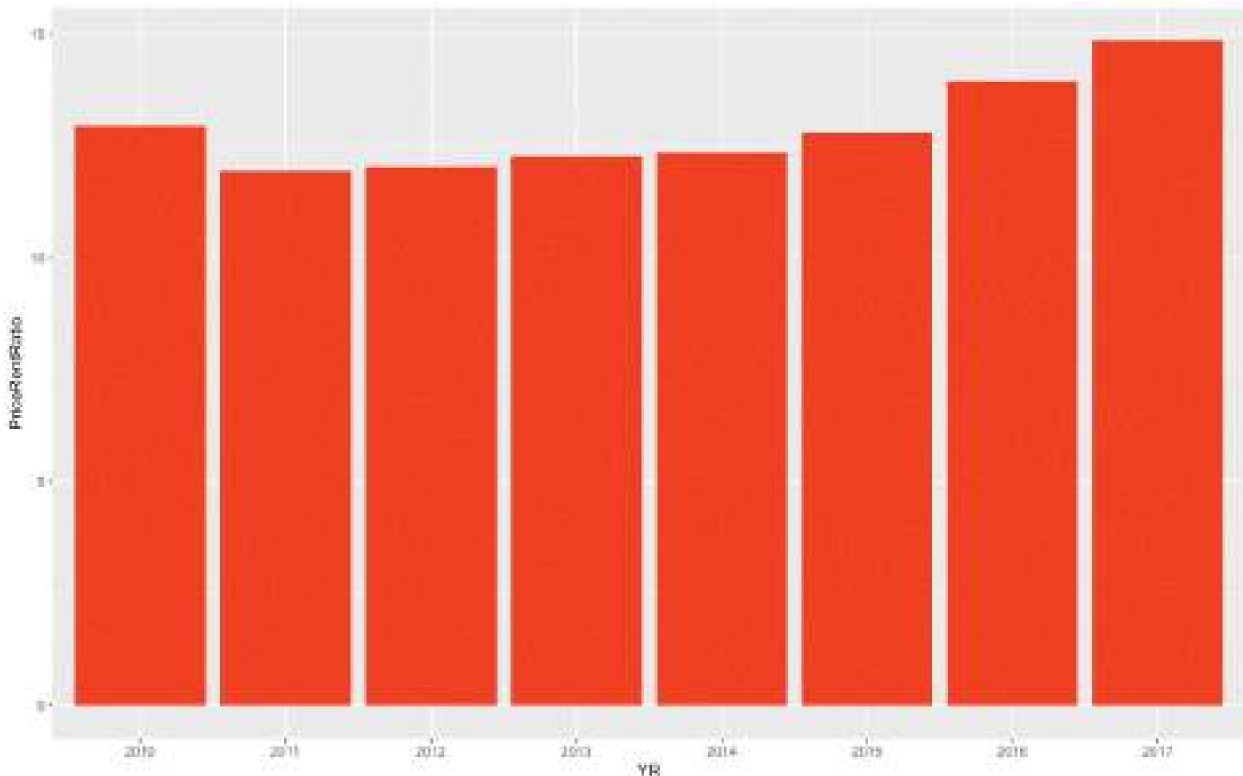
```
dfHomeVals <- select(dfHomeVals, RegionName, State, Metro,`2010` = 12`,
`2014` = `2014-12`, `2015` = `2015-12`, `2016` = `2016-12`, `2017` = `2017-
12`)
dfHomeVals <- filter(dfHomeVals, State == ‘TX’ & Metro == ‘Austin’ &
RegionName == ‘Travis’)
dfHomeVals <- gather(dfHomeVals, `2010`, `2011`, `2012`, `2013`, `2014`,
`2015`, `2016`, `2017`,key=’YR’, value=’ZHVI’)

dfRentVals <- read_csv(“County_Zri_SingleFamilyResidenceRental.
csv”,col_names = TRUE)
dfRentVals <- select(dfRentVals, RegionName, State, Metro, `2010` = `2010-
12`, `2011` = `2011-12`, `2012` = `2012-12`, `2013` = = `2010-12`, `2011` =
`2011-12`, `2012` = `2012-12`, `2013` = 12`, `2017` = `2017-12`)
dfRentVals <- filter(dfRentVals, State == ‘TX’ & Metro == ‘Austin’ &
RegionName == ‘Travis’)
dfRentVals <- gather(dfRentVals, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`,
`2016`, `2017`,key=’YR’, value=’ZRI’)

#join the two df
df <- inner_join(dfHomeVals, dfRentVals, by = ‘YR’)
df <- mutate(df, PriceRentRatio = ZHVI / (12 * ZRI))
ggplot(data=df) + geom_col(mapping = aes(x=YR, y=PriceRentRatio),
fill=”red”)
```

11. You can also check your work against the solution file
CS2_Exercise3.R
.

12. Save the script and then run it to see the output shown in the screenshot below. Price-rent ratios have been steadily increasing during the current decade.



Exercise 4: Comparing residential home values in Austin to other Texas and U.S. metropolitan areas

In this exercise we'll compare residential home values from the Austin metropolitan area to other large metropolitan areas in Texas including San Antonio, Dallas, and Houston. For this exercise we'll create a box plot contained within a violin plot.

1. In RStudio select
File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS2_Exercise4.R
. 2. At the top of the script, load the packages that will be used in this exercise.

library(readr) library(dplyr) library(ggplot2)

3. Use the
read_csv() function from the readr
package to load the data into a data frame.
df <- read_csv("County_Zhvi_SingleFamilyResidence.csv",col_names = TRUE)

4. Select the columns and filter the data. This dataset contains data from 2010 going forward. We’ll use data from December of the years 2010 to 2017 for the Austin, TX metropolitan area.

```
dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
dfHomeVals %>%

select(RegionName, State, Metro,`2010` = `2010-12`, `2011` = `2011-12`,
`2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-
12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
```

5. Filter the data frame to include only Austin, San Antonio, Dallas-Fort Worth, and Houston. These are the four major metropolitan areas in the state.

```
dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
dfHomeVals %>%

select(RegionName, State, Metro,`2010` = `2010-12`, `2011` = `2011-12`,
`2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-
12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
filter(State == ‘TX’ & Metro %in% c(“Austin”, “San Antonio”, “Dallas-
Fort Worth”, “Houston”)) %>%
```

6. Gather the data frame.

```
dfHomeVals %>%
select(RegionName, State, Metro,`2010` = `2010-12`, `2011` = `2011-12`,
`2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-
12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
filter(State == ‘TX’ & Metro %in% c(“Austin”, “San Antonio”, “Dallas-Fort
Worth”, “Houston”)) %>%
gather(`2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`,
`2017`,key=’YR’, value=’ZHVI’) %>%
```

7. Group the data by metropolitan area.

```
dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
```

dfHomeVals %>%

select(RegionName, State, Metro,`2010` = `2010-12`, `2011` = `2011-12`, `2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
filter(State == 'TX' & Metro %in% c("Austin", "San Antonio", "Dallas-Fort Worth", "Houston")) %>%
gather(`2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`,key='YR', value='ZHVI') %>%
group_by(Metro) %>%

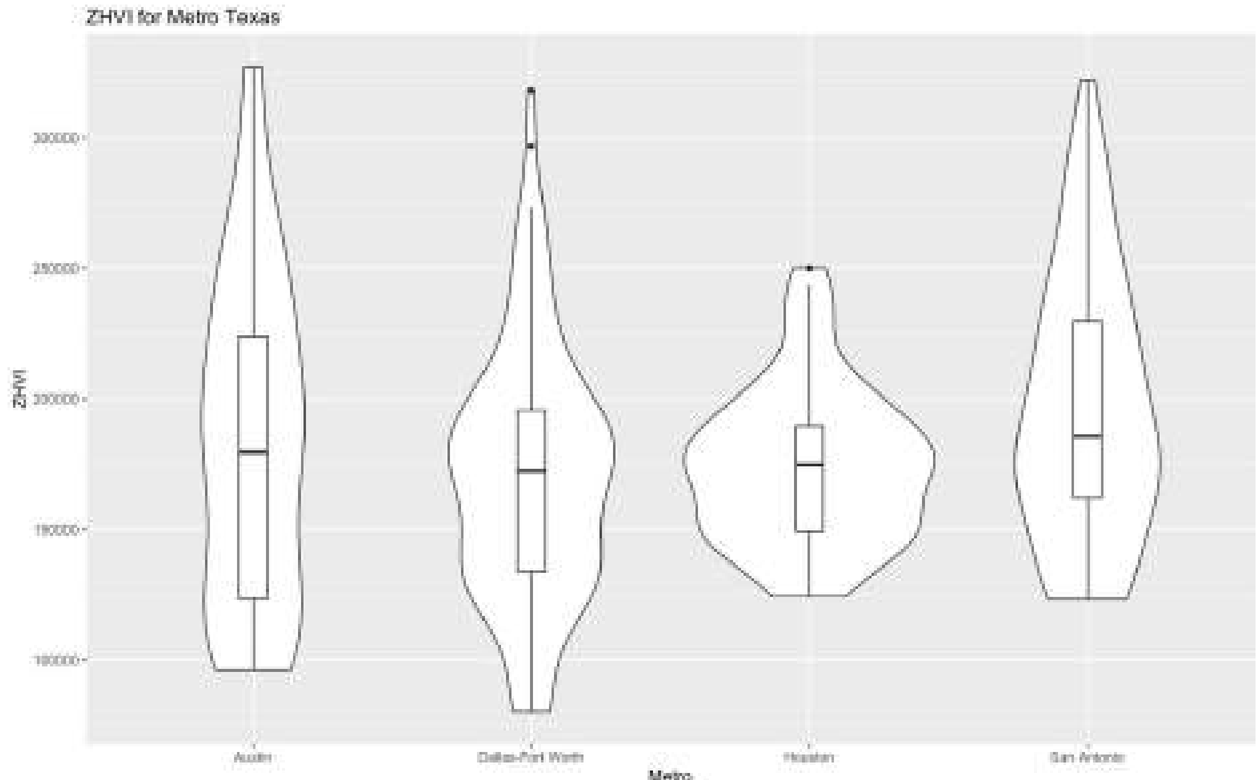
8. Use
ggplot() with geom_violin() and geom_boxplot()
to create the plot.

dfHomeVals <- read_csv("County_Zhvi_SingleFamilyResidence.
csv",col_names = TRUE)
dfHomeVals %>%

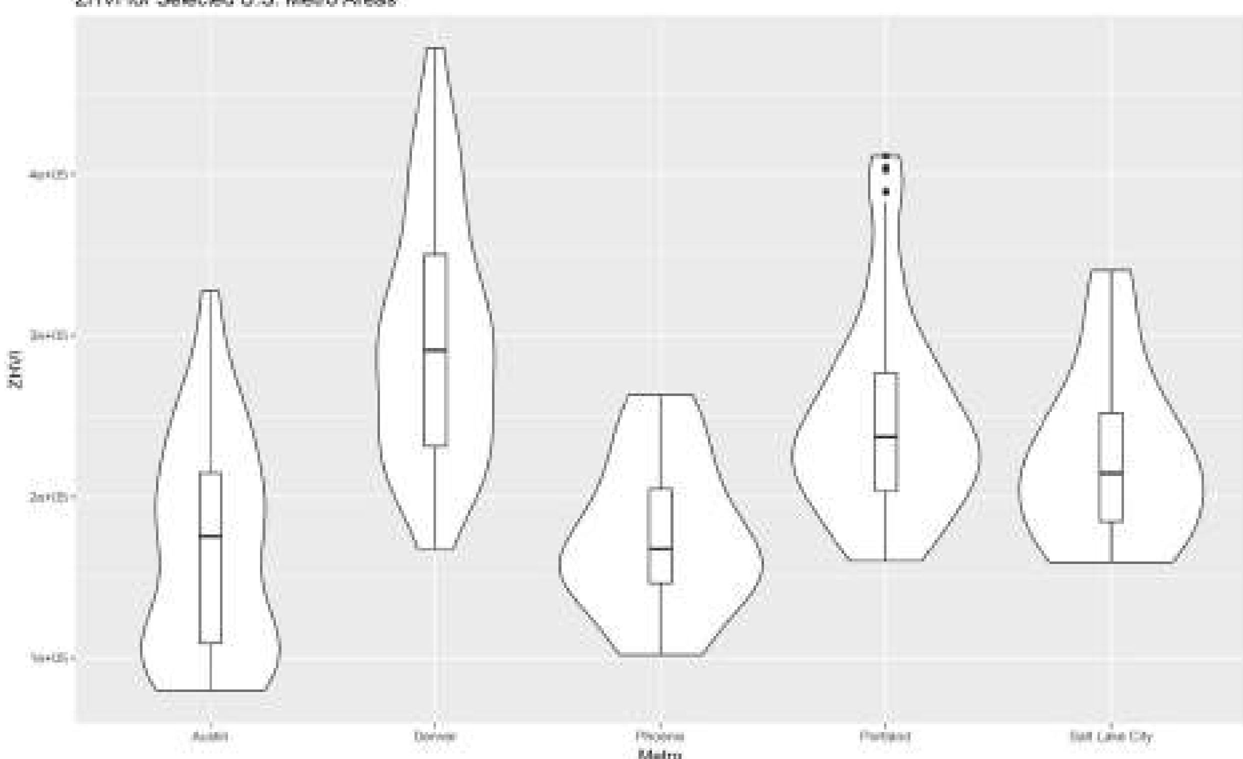
select(RegionName, State, Metro,`2010` = `2010-12`, `2011` = `2011-12`, `2012` = `2012-12`, `2013` = `2013-12`, `2014` = `2014-12`, `2015` = `2015-12`, `2016` = `2016-12`, `2017` = `2017-12`) %>%
filter(State == 'TX' & Metro %in% c("Austin", "San Antonio", "Dallas-Fort Worth", "Houston")) %>%
gather(`2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`,key='YR', value='ZHVI') %>%
group_by(Metro) %>%
ggplot(mapping = aes(x=Metro, y=ZHVI)) + geom_violin() + geom_boxplot(width=0.1) + ggtitle("ZHVI for Metro Texas") + xlab("Metro") + ylab("ZHVI")

9. You can also check your work against the solution file
CS2_Exercise4.R
.

10. Save the script and then run it to see the output shown in the screenshot below.



11. Challenge: Update the script to include the following metropolitan areas: Austin, Denver, Phoenix, Salt Lake City, Boise, Portland. You can check your code against the solution file CS2_Exercise4.R. The output plot should appear as seen in the screenshot below.



12. Finally, we’ll create a script that displays the ZHVI values for each metropolitan area in a facet plot. In RStudio select File | New File | R Script and then save the file to the CaseStudy1 folder with a name of CS2_Exercise4B.R.

13. At the top of the script, load the packages that will be used in this exercise.

```
library(readr)
library(dplyr)
library(ggplot2)
```

14. Use the
read_csv() function from the readr
package to load the data into a data frame.

```
df <- read_csv(“County_Zhvi_SingleFamilyResidence.csv”,col_names = TRUE)
```

15. Define the columns to use. In this case we’ll use the years 2000-2017.

```
dfHomeVals %>%
select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =
select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =
select(RegionName, State, Metro, `2000` = `2000-05`, `2001` = 05`, `2008` =
`2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012`
```

= `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`,
`2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>%

16. Filter the data frame to include only specific metropolitan areas.

dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
dfHomeVals %>%

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` = 05`, `2008` =
`2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012`
= `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`,
`2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>%
**filter(Metro %in% c(“Austin”, “Denver”, “Phoenix”, “Portland”, “Salt
Lake City”)) %>%**

17. Gather the data.

dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
dfHomeVals %>%

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` = 05`, `2008` =
`2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012`
= `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`,
`2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>%
filter(Metro %in% c(“Austin”, “Denver”, “Phoenix”, “Portland”, “Salt Lake
City”)) %>%
**gather(`2000`, `2001`, `2002`, `2003`, `2004`, `2005`, `2006`, `2007`, `2008`,
`2009`, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`,
`2017`,key=’YR’, value=’ZHVI’) %>%**

18. Group the data by metropolitan area.

dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
dfHomeVals %>%

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` = 05`, `2008` = `2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012` = `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`, `2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>%
filter(Metro %in% c(“Austin”, “Denver”, “Phoenix”, “Portland”, “Salt Lake City”)) %>%
gather(`2000`, `2001`, `2002`, `2003`, `2004`, `2005`, `2006`, `2007`, `2008`, `2009`, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`,key=’YR’, value=’ZHVI’) %>%
group_by(Metro) %>%

19. Plot the data as a facet plot.

dfHomeVals <- read_csv(“County_Zhvi_SingleFamilyResidence.
csv”,col_names = TRUE)
dfHomeVals %>%

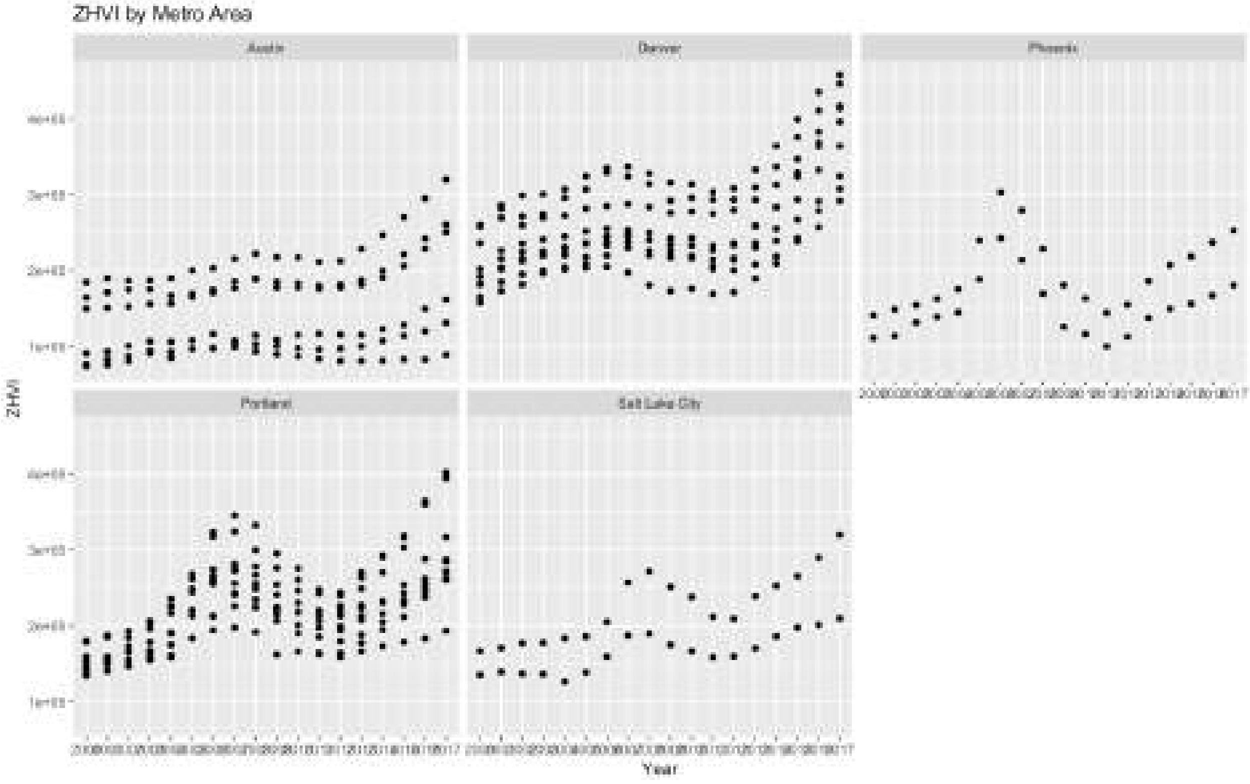
select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` =

select(RegionName, State, Metro, `2000` = `2000-05`, `2001` = 05`, `2008` = `2008-05`, `2009` = `2009-05`, `2010` = `2010-05`, `2011` = `2011-05`, `2012` = `2012-05`, `2013` = `2013-05`, `2014` = `2014-05`, `2015` = `2015-05`, `2016` = `2016-05`, `2017` = `2017-05`, `2018` = `2018-05`) %>%
filter(Metro %in% c(“Austin”, “Denver”, “Phoenix”, “Portland”, “Salt Lake City”)) %>%
gather(`2000`, `2001`, `2002`, `2003`, `2004`, `2005`, `2006`, `2007`, `2008`, `2009`, `2010`, `2011`, `2012`, `2013`, `2014`, `2015`, `2016`, `2017`,key=’YR’, value=’ZHVI’) %>%

group_by(Metro) %>%
ggplot(mapping = aes(x=YR, y=ZHVI)) + geom_point() + facet_wrap(~Metro) + geom_smooth(method=lm, se=TRUE) + ggtitle(“ZHVI by Metro Area”) + xlab(“Year”) + ylab(“ZHVI”)

20. You can also check your work against the solution file CS2_ Exercise4B.R.
21. Save the script and then run it to see the output shown in the screenshot below.



Data Visualization and Exploration with R

Today, data science is an indispensable tool for any organization, allowing for the analysis and optimization of decisions and strategy. R has become the preferred software for data science, thanks to its open source nature, simplicity, applicability to data analysis, and the abundance of libraries for any type of algorithm.

This book will allow the student to learn, in detail, the fundamentals of the R language and additionally master some of the most efficient libraries for data visualization in chart, graph, and map formats. The reader will learn the language and applications through examples and practice. No prior programming skills are required.

We begin with the installation and con figuration of the R environment through RStudio. As you progress through the exercises in this hands-on book you'll become thoroughly acquainted with R's features and the popular tidyverse package. With this book, you will learn about the basic concepts of R programming, work efficiently with **graphs,** charts, and maps, and create publication-ready documents using real world data. The detailed step-by-step instructions will enable you to get a clean set of data, produce engaging visualizations, and create reports for the results.

What you will learn how to do in this book:

Introduction to the R programming language and R Studio

Using the tidyverse package for data loading, transformation, and visualization

Get a tour of the most important data structures in R

**Learn techniques for importing data, manipulating data,
performing analysis, and
producing useful data
visualization**

Data visualization techniques with ggplot2

Geographic visualization and maps with ggmap

Turning your analyses into high quality documents, reports, and presentations with R Markdown.

Hands on case studies designed to replicate real world projects and reinforce the knowledge you learn in the book

For more information visit *geospatialtraining.com*!