Data Science with R

by Matthew Renze

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hi, I'm Matthew Renze with Pluralsight, and welcome to Data Science with R. Data science is the practice of transforming data into knowledge, and R is the most popular open-source programming language used for data science. In our data-driven economy, this combination of skills is in extremely high demand, commanding significant increases in salary as it's revolutionizing the world around us. In this course, you'll learn how to use the principles of data science and the R programming language to answer day-to-day questions about your data. As an overview of this course, first, we'll learn about the practice of data science and the R programming language. Then, we'll learn how to work with data to create descriptive statistics, data visualizations, and statistical models. Finally, we'll learn how to handle big data, make predictions with machine learning, and deploy our applications into production. By the end of this course, you'll have the skills necessary to use R and the principles of data science to transform your data into actionable insight. In addition, by the end of this course you'll have built and deployed a web-based interactive machine learning application that allows users to make predictions using data. As an introductory course, there are no prerequisites for this course. However, having basic experience with at least one programming language and basic knowledge

of statistics will be beneficial. So please join us today at Pluralsight and learn how to transform your data into actionable insight using data science with R.

Introduction to Data Science

Introduction

Hi, my name is Matthew Renze with Pluralsight, and welcome to Data Science with R. In this introductory course, we'll learn about the statistical programming language R, and how we can use it to perform data science in order to transform our data into actionable insight. When you're finished with this course, you'll have the skills necessary to answer day-to-day questions about your data using R. So let's get started. Whether you realize it or not, there's a flood of data coming our way. In fact, the flood is already here and it's growing exponentially every year. Over the next few years, we're going to see two entirely different outcomes for individuals, businesses, and governments, based on whether they learn to use these data to their advantage or not. Either these people, businesses, and governments are going to sink in the coming sea of data, or they're going to learn how to swim. So I'm currently doing what I can in my professional career to learn how to swim in this new data-driven economy, and I encourage you to do the same. Essentially, in and information economy, data is the new oil. And much like crude oil must be refined into fuel to be of value to us, raw data must also be transformed into knowledge to be of value to us as well. I think Google's chief economist summed the idea up very well when he said, The ability to take data -- to be able to understand it, to process it, to extract value from it, to visualize it, to communicate it -- that's going to be a hugely important skill in the next decades, because now we really do have essentially free and ubiquitous data, so the complimentary scarce factor is the ability to understand that data and extract value from it. And we see the same sentiment expressed by experts across various industries. In fact, I see articles like these popping up on a daily basis these days. The reality is that there's an extremely high demand for IT professionals with the ability to transform data in to actionable insight. However, there are nowhere near enough people available with the proper skills and tools to meet this demand. This means that people with the right set of skills are commanding significantly higher salaries in the IT industry right now. In addition, of all the tools available for transforming data into knowledge, the programming language R is consistently one of the top-most used and demanded tools. So I'm glad you've decided to join the data scientists, data analysts, and developers who have learned

this valuable set of skills. In addition, I'm very happy to teach you a bit of what I know about data science with R.

Course Overview

As an overview of this course, first we'll learn about data science, the data-science process, and why it has become so important in recent years. Next, we'll learn about the R programming language, and we'll learn how to program using R. Then, we'll learn how to work with data in order to import, clean, transform, and export data. Next, we'll learn how to analyze our data numerically by creating descriptive statistics. Then, we'll learn how to analyze your data visually by creating data visualizations. Next, we'll learn how to create statistical models to make statistical inferences and predictions using our data. Then, we'll learn how to handle big data scenarios in R, that is, data sets that are too large to process via traditional means. Next, we'll learn how to train machine-learning models to make predictions based on our data. Finally, we'll learn how to deploy R into production using a web-based, interactive application framework called Shiny. So by the end of this course, you'll have built a web-based, interactive, machine-learning application in R that can take input from a user and make predictions as output. The purpose of this course is to act as a high-level introduction to the most important topics in data science using R. We won't go too deep into any one specific topic; rather we'll just learn the most important concepts and defer a much more in-depth coverage of each specific topic to other Pluralsight courses and other sources of information. There are several course on Pluralsight that provide much more in-depth coverage of each of the main topics we'll cover in this course. So I encourage you to use this course as a high-level guide to get you started with data science using R, and then use the more focused courses for more in-depth training on areas that you are most interested in learning more about. I'll point you in the direction of all the other courses that currently exist on the Pluralsight website, and the topics that they expand upon at the end of the last module of this course. The intended audience for this course are developers who want to learn how to use R in the data science process to build data-driven applications, data analysts who want to learn how to use R to perform their data analyses, and IT professionals who want to learn how to make better decisions using data. Essentially, this course is for anyone in information technology with a desire to transform their data into actionable insights using R. Since this is an introductory course, there's no required perquisites. However, having basic programming experience with at least one programming language will help you understand the programming concepts more easily. In addition, having a basic knowledge of statistics will be beneficial as well.

However, if you do not meet either of these two optional prerequisites, you should be able to follow along just fine. So now, let's learn about data science and the data science process.

Introduction to Data Science

The first question we need to answer is what is data science? In essence, data science is an interdisciplinary field composed of computer science, math and statistics, and domain knowledge, that is subject matter expertise that seeks to derive insight from data. In other words, the goal of data science is to transform data into knowledge, knowledge that can be used to make rational decisions that lead to beneficial outcomes. The Venn diagram on the screen shows how the three disciplines combine to create the interdisciplinary field of data science. So what, then, is a data scientist? A data scientist is a person who performs data science for a living. However, a data scientist is more than just a scientist, more than just a data analyst, and more than just a developer; they possess all three sets of skills. Individuals with all three sets of skills and proper credentials are currently very rare in our industry. Despite how rare actual data scientists are, they are currently in very high demand. In addition, that demand is growing every year as we can see by the trend of job postings on Indeed. com looking for data scientists. So there's clearly a high demand for this specific set of skills, whether you are a data scientist or just applying the principles of data science to your current job role. The rarity of data scientists, combined with their high demand, leads to much higher salaries that the data scientists and individuals with similar skills can command. It's also important to note that you'll not magically become a data scientist after finishing this course. However, this course will teach you basic skills of data science that you can begin applying at your job today, and will point you in the right direction if a career in data science is something you're interested in learning more about. So what set of skills does a data scientist possess? In general, the skills of data science are, programming computers using a programming language like R, working with data, that is collecting, cleaning, and transforming data, creating and interpreting descriptive statistics, that is numerically analyzing data, creating and interpreting data visualizations, that is visually analyzing data, creating statistical models and using them for statistical inference, hypothesis testing, and prediction, handling big data, that is data sets that are too large to work with via conventional means, making predictions using machine-learning algorithms, and deploying their work into production environments or communicating their results to a wider audience. If you recall from the outline of this course we saw earlier, we're going to be touching upon each of these skills throughout the remaining modules of this course. Regarding the tools used in data science, here are the tools listed by the respondents of O'Reilly's Data Science Salary Survey in 2015. On the X axis we have the name of

the tool, which includes programming languages, data platforms or analytics tools. On the Y axis, we have the percent of respondents reporting that they used that corresponding tool. As we can see, R is listed as one of the most popular tools used for data science. In addition, across a variety of similar surveys I've looked at, we see the same four tools at the top of most surveys. Typically, we see SQL first, followed by Excel, and then both R and Python are usually pretty neck in neck in terms of their popularity. So why has data science become so important all of a sudden? There are a few trends really driving the importance of data science. First, emerging trends like the internet of things are creating massive amounts of relatively low-cost data. This means that we now have access to large amounts of valuable, low-cost, near real-time data. Second, emerging trends like big data are creating new tools like Hadoop that allow us to process big data sets at scale. This means that we can process these massive data sets with relatively low cost, and in a reasonable amount of time. Third, we have powerful machine-learning tools that can find patterns and make predictions with these data. Emerging trends like deep learning via deep neural networks are making machines significantly better at making predictions than humans in certain categories of decision-making. Finally, in the past we would collect data and analyze it so that human beings could use the results of the analysis to make a decision and take action. However, we're now closing the loop of the internet of things, big data, and machine learning, so that human beings are no longer required in the decision-making process. This leads us to fully autonomous systems in environments that function without the need for a human decision maker in the process. These fully autonomous systems will transform our world in more ways than I could possibly enumerate. Self-driving cars, intelligent buildings, and smart cities are either already here, or just around the corner. In fact, it's a very exciting time to be learning about data science. Essentially, the recent importance of data science all boils down to two things. It's being driven by economics and it's being made possible by technology. The cost of collecting, storing, processing, and analyzing data is continuously decreasing. And the value that these data can provide when the data science process is applied correctly continues to increase. In addition, we now have the technological capabilities to close the loop to create fully autonomous systems. The cost savings and value-generating possibilities of these fully autonomous systems are very compelling. Given this, it's unlikely that the demand for data science will decrease anytime in the near future. The data science process works like this: First, we find a question that we want to answer. This can be a hypothesis we want to test, a decision we want to decide on, or a prediction we want to make. Second, we explore the data that we have available and the data that we may need to collect. We need to understand what data we have, what data we don't have, and what condition these data are in. Third, we prepare the data for analysis, a process often referred to as data munging or data wrangling. We need to gather, clean, and transform

these data to get them into a form suitable for analysis. Fourth, we create a model given our data. In the most generic sense, this can be a numerical model, a visual model, a statistical model, or a machine-learning model. Fifth, we evaluate the model. We need to determine if our model answers our question, helps us make an informed decision, or creates an accurate prediction. In addition, we also need to make sure that the model is appropriate, given our data and the context. Finally, if everything looks good, we deploy our model. This could mean either communicating the results of our analysis to others, making a decision, or implementing an application in production. Then, we repeat this process for each question we'd like to answer. This is typically an iterative process. We typically go through the complete cycle multiple times, learning and improving with each iteration. In addition, this process is often non-sequential. We often have to bounce back and forth between steps as we discover problems and learn better ways of solving these problems. Sometimes we don't always need to complete the process. Often, we learn that what we're doing isn't working or doesn't make sense given the data or the context, so we terminate the process and shift our focus to the next most important data-science question. There are also some well-established practices for the data science process available, like the CRISP-DM process, which stands for Cross Industry Standard Process for Data Mining. These established practices are useful to help you get started with your data science process.

Summary

In this module, first we introduced the topic of this course and explained why we want to learn about data science and R. We discussed the factors that are driving the demand for both data science and tools like R. Next, we provided an overview of the course. We discussed the course outline, intended audience, prerequisites, and more. Finally, we introduced the topic of data science. We learned what data science is, why is has become so important, and how the data science process works. In the next module, we'll learn about the R programming language and how to use it for data science.

Introduction to R

Introduction

Hello, and welcome back to Data Science with R. I'm Matthew Renze with Pluralsight, and in this module, we'll learn about the programming language R. As an overview of this module, first, we'll

learn about the R programming language, what it is, and why it has become so popular. We'll also learn about the RStudio integrated development environment, or IDE. Then, we'll see a demo of the basic syntax of the R language, we'll learn about data types, data structures, and how to slice and dice data in R. So let's get started.

Introduction to R

The first question we need to address is what is R? R is a free, open-source implementation of the statistical programming language S. S was a statistical programming language invented in Bell Labs in 1976. So R is the free open-source implementation of that language. R is both a programming language and an environment within which to program. That is, we have a programming language called R, and a development environment for that language, which are both referred to as R. It provides both numerical and graphical data analysis capabilities. That is, we can analyze our data statistically, or we can analyze data visually by creating data visualizations. It's also cross platform, which means it runs on Windows, Mac, and UNIX systems. In addition, R is actively under development, has a large user community, is very modular and extensible, and has over 9, 000 extension packages. And best of all, R is free open-source software, meaning that it's free, as in beer, meaning anyone can use the software without cost, and free, as in speech, meaning anyone can view the source code, learn from it, modify it, or redistribute it as they wish. As I mentioned earlier, R is a very popular programming language. So, in order to show where it lies in terms of programming language popularity, here's a data visualization from RedMonk showing us where R lies relative to other programming languages. On the X axis, we have programming language Popularity on GitHub by # of Projects. On the Y axis, we have the Popularity Rank on Stack Overflow by # of Tags. As we can see at the upper right hand corner, we have the very popular, general-purpose programming languages. For example C#, Java, and Python. In this next cluster, we have the popular, more specialized languages, for example MatLab, Lisp, and Erlang. And finally, down here we have the less popular languages. Here we have R, right at the top of the more popular, specialized languages. In addition, I should also note that R will most likely never be as popular as the general-purpose languages. It's specialized to work with data extremely well, but you probably won't be using it to build your next E-Commerce website anytime soon. This is what R looks like. As you can see, we have both a programming language and an environment within which to program. In addition, we can see that R can do both numerical analysis and graphical analysis as well. Rather than working with the out-of-the-box R IDE for the demos in this course, we're going to be using a more powerful, integrated development environment, or IDE, called RStudio. It still uses the R

programming language, however, the IDE is much more user friendly and makes it much easier to do code demos. We'll walk through the various windows in RStudio during the first code demo, which is coming up shortly. In addition, I think it's important that I should point out that there's now support for R in Microsoft Visual Studio using R tools for Visual Studio, or RTVS, for short. If you're a software developer that uses Visual Studio as your primary IDE, you might prefer to use R in Visual Studio instead. There are pros and cons to each of these three IDEs; however, I'll leave it up to you to try each of them out and choose which one you're most comfortable with. Before we begin our demos, we need to download and install both R and RStudio. You can download R from the R Project website at www. r-project. org. Downloading and installing R on your computer is relatively easy and straightforward, so we won't cover these steps in our upcoming demo. Ultimately, I don't think you should have any difficulties getting R up and running if you follow along with the instructions included. Next, we need to download and install RStudio. You can download RStudio from the RStudio website at www. rstudio. com. Once again, the installation is easy and pretty straightforward, so we'll pause here to give you a chance to get your computer set up in the event you'd like to follow along, and then we'll see a demo of the R language basics next.

Demo

For our first demo, we're going to complete a Hello World example using the R programming language inside of the native R development environment. After this demo, we'll be using RStudio going forward for the remaining demos in this course. To perform Hello World, we want to set a variable x equal to the character string Hello World, and then print this variable to the Console window. First, we type into the console x, followed by the left arrow assignment operator, then Hello World surrounded in double quotes. When we press Enter, the character string Hello World is stored in a variable named x. Next, we type the print function and pass x in as an argument to the function. When we press Enter, the value that was assigned to the variable x, that is Hello World, will be output to the Console window. Successfully print Hello World to the console using a variable in R. So now let's --- native R development environment to RStudio to make things easier for the remainder of our demos. This is the RStudio integrated environment, or IDE, for short. There are four main windowpanes in RStudio. First, in the upper left hand corner we have the Source pane. It contains the scripts that we'll be running for each of our demos. Next, in the lower left hand corner we have the Console pane. This is where we can interactively program in R. We type commands and output appears in the Console window. Then, in the upper right hand corner, we have the Environment pane. The Environment pane shows us the state of all the

variables that are currently in memory. Finally, in the lower left-hand corner we have the Miscellaneous pane. Here we find tabs for navigating the file system, viewing charts and graphs, and reading the help documentation. In order to make the code demos run more smoothly, we're going to be executing scripts from the Scripts pane line by line, rather than typing everything by hand. We could do everything by hand, like in the Hello World demo, but typing during demos is error prone and doesn't help move the demos along any faster. It'll work like this. I'll place my cursor on the line of script I plan to execute and I'll press Ctrl+Enter, which will execute that line of code in the console. You'll see both the command being executed and the results of the command in the Console window. To see how this will work, let's just walk through the Hello World demo again in RStudio. First, in the console we'll set x to the character string, Hello World. Next in the Environment pane, we'll see that the variable x has been assigned to the value Hello World. Then in the console we'll print x, which writes Hello World to the Console window. Easy enough? Excellent! Now let's take a look at the basic-language features in R. As we saw in the Hello World demo, we can assign a value to a variable using the left arrow assignment operator. This operator assigns the value on the right to the variable on the left. In addition to the left arrow assignment operator, there are two other value-assignment operators in R. The second assignment operator is the equal-sign assignment operator. For the most part, the equals sign assignment operator behaves similar to the left arrow assignment operator. However, there is a slight technical difference regarding the scope of the variable's assignment. This technical difference is a bit difficult to explain, especially if you don't have a background in programming, so we'll defer an explanation for now. As a general rule for this course, however, always use the left arrow assignment operator unless you see me using the equal sign assignment operator, which we'll do from time to time, for example, when creating columns for a data frame. There's also a third assignment operator called the right-arrow assignment operator. This operator assigns the value on the left to the variable on the right. However, this operator is rarely, if ever used, so I recommend avoiding it in general. For all of our demos in this course, we'll be using the left-arrow assignment operator exclusively, except in cases where we specifically need to use the equal-sign assignment operator. R has a language feature called implicit printing. Rather than having to type print x every time we want to write the value of a variable x to the console, which we refer to as explicit printing, we can implicitly print the variable instead. Implicit printing allows us to simply type the name of the variable, for example x, and when we press Enter, the value of the variable is automatically printed to the console. We'll be using implicit printing for the remainder of this demo. However, we'll use explicit printing in later demos to make it more clear what is occurring during those demos. To create variables in R, we just assign a value to a variable. For example, we can create a logical, that is a Boolean variable, and set its value to either

true or false. We can create an integer variable by setting its value to an integer and using the capital letter L suffix. The L stands for long, which is programming language shorthand for long integer, which is a 32-bit integer. Integers are either positive or negative, whole number values. We can create a numeric variable by setting its value to a numeric constant. Numeric variables contain decimal precision numeric values. We can create a character string by setting its value to a string of characters surrounded by double quotes. The character data type can store an arbitrary length string of characters, and we can create date and time data types in R, as well. For example, to create a date representing February 3, 2001, we use the as. Date operator and pass in a date in one of the many date/time formats recognized by R. There are a few other primitive data types as well. For example, data types for working with both dates and times together and for dealing with complex numbers. But these five data types are the most frequently used ones in the language. In addition, these are the only data types that we'll be using for the demos in this course. Once we have created variables, we can display the value of these variables in the console by either explicitly or implicitly printing them. For example, our logical variable prints the value TRUE, as we'd expect, our Integer variable prints a whole number, our numeric variable prints a decimal precision numeric value, our character vector prints out a string of characters, and our date data type prints out a date in the ISO8601 International Standard data format, that is, year, month, and day, separated by dashes. R is a functional programming language, so functions are first-class citizens in R. This means that we can assign functions to a variable and pass them around just like variables containing data. We create a function by using the function keyword and defining the arguments in the body of the function. For example, to create a function f with a single argument x in a body that returns x + 1, we type f, set equal to a function, with the argument x in a body that returns x + the number 1. Now, if we want to invoke that function, we simply type the name of the function f and pass in an argument for x. In this case, we passed in the value 2, so 2 + 1 returns 3. In R we can assign functions to named variables or we can use anonymous functions as well. That is, we can use a function without first assigning it to a named variable. In addition to primitive data types and functions, we have more complex data structures in R as well. Data structures are containers that hold a collection of data in one of several ways. The most important of these data structures in R is a vector, which is essentially a one-dimensional array containing elements of the same data type. To create a vector, we use the concatenation function c, and pass in a comma-separated list of homogenous values, that is values that are all of the same data type. This function combines each of the individual elements into a vector containing those elements. For example, to create a numeric vector containing the values 1, 2, and 3, we set v equal to a vector containing the values 1, 2, and 3. When we print the variable v, we see a vector containing the values 1, 2, and 3. We can also create a vector of

numerically ascending or descending values using the sequence operator, which is the colon character. For example, to create a vector containing the values 1 through 5, we set a variable s equal to the values 1 through 5. When we print this variable s, we see a vector containing the numeric values 1 through 5 in ascending order. Next, we can create a matrix, which is a special case of a two-dimensional array using the matrix keyword. For example, if we want to create a 2 x 3 matric containing the values 1 through 6, populated in columnar fashion, we'd set a variable m equal to a matrix containing the sequence of data 1 through 6 with 2 rows and 3 columns. When we print this variable m, we see a 2 x 3 matrix containing the values 1 through 6, populated in columnar fashion. We can create multi-dimensional arrays in R using the array keyword. Arrays can have 1, 2, or more dimensions. For example, if we want to create a 2 x 2 x 2 array, that is, a 3-dimentional array containing the values 1 through 8, populated in columnar fashion, we would set a variable a equal to an array with the data 1 through 8 and set the dimensions argument equal to a vector containing the values 2, 2, and 2. When we print the variable a, we see a 2 x 2 x 2 array containing the values 1 through 8, populated in columnar fashion. In addition to creating data structures with homogenous data types, that is data structures that contain values of all of the same data types, like vectors, matrices, and arrays, R also contains a special data structure called a list for storing collections of heterogeneous data types, that is a list of values of different data types. For example, to create a list of heterogeneous values in R, we would set a variable l equal to a list containing the logical value true, the integer 123, the numeric value 2. 34, and the character string abc. When we print the variable l, we see a list containing the values TRUE, the integer 123, the numeric value 2. 34, and the character string abc. The formatting of the output of the list in the R Console might look a bit messy and difficult to read, however, this formatting has to do with the way R stores the values contained in the list within memory. There's a special data type in R called a factor that stores categories of named values. This is similar to an enumeration in the C-like programming languages. Vectors store a list of categorical variables by assigning each unique category, called a level, as an integer value and storing the data as a vector of integers in memory, instead of storing the entire list of categorical values. Doing so saves a significant amount of space in memory and affords significantly greater performance for many operations. For example, if we have a vector of character strings containing the named values of Male, Female, Male, Male, Female, we can store these values using a factor. First we set a variable called factor, equal to the factor keyword given a vector of character strings. If we print this factor, we can see that it contains five values, Male, Female, Male, Male, Female, and two levels, that is Female and Male. If we use the levels command, we'll see a list of unique-name values or levels in alphabetical order. The order in which these named values occur is the order that maps to the backing integers. For example, Female maps to the integer 1, and Male maps to the integer

2. If we use the unclass command, we can see the underlying integer array that is 2 1 2 2 1 and the 2 levels, that is Female and Male that compose this factor. We'll be using factors to store all of the categorical values that we'll be working with in this course. A data frame is the most important data structure for working with collections of tabular data in R. It essentially represents a table of data, that is, it has a set of columns and a set of rows. Each column can contain a different data type, but all the data in a single column must be of the same data type. We create a data frame using the data frame keyword. For example, we can set a variable df equal to a new data frame containing a column called Name, populated with a vector of character string values, Cat, Dog, Cow, and Pig, and a second column called HowMany, containing a vector of integers, 5, 10, 15, and 20, and a third column called IsPet containing a vector of logical values TRUE, TRUE, FALSE, and FALSE. When we print our data frame, we get pretty much what we'd expect, a table containing three columns, that is, Name, HowMany, and IsPet, in four rows containing their respective values. Once we have our data stored in a data frame, we can query and manipulate these data in extremely powerful ways. We can index our data frame by row and column using the index operator. For example, if we want to return the value contained in row 1 in column 2, we would type the following: As we can see, this returns the integer 5, which is the value contained in row 1 in column 2 of our data frame. In addition, we can omit the column argument and the index operation will return the values in the first row across all columns. Or, we can omit the row argument and the indexing operation returns all values contained in the second column. We can also index by columns using the column name with the double square brackets notation. Or, if there's a syntactic shortcut that allows us to use the dollar sign followed by the name of the column. All three of these examples return the same values, that is, all rows in the second column. However, the last of these three methods is generally preferred. One of the most powerful language features for querying data in a data frame is called Subsetting. Subsetting allows us to slice and dice data in very flexible ways. First, we can pass a vector of integers indicating each row we want returned from our data frame into the rows argument of the index operator and it will return the specified rows. For example, passing in a vector containing the integers 2 and 4 will return rows 2 and 4. In addition, we can pass a sequence of integers, for example, rows 2 through 4 into the rows argument and it will return rows 2 through 4. We can also pass in a vector of logical values, for example, TRUE, FALSE, TRUE, and FALSE, indicating which rows we want returned or not, and it will return the corresponding rows. In this case, row 1 and row 3, since the first and third elements in the vector were set to TRUE. Although passing a vector of logical values by hand might seem like a trivial example, you can imagine how powerful this would be if the values in the vector were created programmatically, using an operation that returns true or false for each row in the database. For example, we can use the equality operator to return TRUE

or FALSE for each row in the database matching a certain condition. As an example, we could specify that we want to return all rows where IsPet is TRUE, and we'll get the rows containing Cat and Dog. We can also subset using inequality operators. For example, we can query all rows where how many is greater than 10, which returns the rows containing Cow and Pig. We can also subset data tables using more advanced query expressions, like finding all rows matching a list of values. For example, we can find all rows where the animal's name is contained in the list Cat or Cow, and this will return just row 1 and row 3. And I should note that indexing and subsetting operations also apply to vectors, matrices, arrays, and lists, in additional to data frames. A very important thing to note about R is that it's a vectorized language. This means that all atomic data types in the language are vectors of size 1. What this means is that every time we assigned a single value to a variable, we aren't actually assigning a single atomic data type to the variable like we would in most other programming languages, instead we're creating a vector of the specified data type in assigning a value to the first element of the vector. Now, this might seem a bit odd at first, because this is very different from most programming languages you've probably worked with. However, it makes perfect sense in R, because we're typically working with collections of data. For example, vectors, matrices, arrays, lists, and data frames. So working with them as collections is often more practical and efficient. For example, if we add the values 1 and 2 we get the value 3, like you would expect, however, this 3 is actually a vector containing a single element with a value of 3. In addition, most operations in R are vectorized, meaning that they take a vector's input and return a vector as a result. For example, if we were to add a vector containing the values 1, 2, and 3, to a vector containing the values 2, 4, and 6, we'd get a resulting vector containing the values 3, 6, and 9. This is because the pairwise addition of 1 and 2 is 3, 2 and 4 is 6, and 3 and 6 is 9. Another important language feature to note is named versus ordered parameters. With R, we can either pass arguments into functions, using the name of the parameter, or we can pass arguments into functions by their default order. We learn the default order of parameters by looking at the help files for each command. For example, earlier, we created a matrix with the data 1 through 6 with the number of rows equal to 2, and the number of columns equal to 3. However, we can also create an identical matrix passing the same values by their default order, provided that we know the correct parameter order, which is available from the help files. Now, if we check to see if the elements of these two matrices are equal, which they should be, we can see that we get a matrix back with the value TRUE in all cells. This is because the pairwise equality of each value in each cell in both of these matrices is TRUE. Notice that the equality operator for matrices is vectorized as well. We gave it two matrices and it returned a matrix as a result, instead of either a single TRUE or FALSE answer. However, if you're interested in testing if these two matrices are indeed identical, that is the scalar equivalent to the vectorized

equality operation, we can see that the operation returns TRUE, as the two matrices are indeed identical. The fact that R is a vectorized language is one of the most interesting and powerful features of R. However, it is one of the most confusing aspects for both developers and non-developers new to R, as it takes a bit of time to get used to. In fact, because of this feature, it is very rare that you use for loops in the language. This is quite different than most programming languages you've probably used where iterating over collections is a very normal part of programming. If you really want to understand the power of R for processing data, it's important to have a sold grasp on this concept. As I mentioned earlier in this module, R is very modular and extensible. It's very easy to download and install packages that contain new functionality that extends the power of the language. For example, if we would like to install an extension package called dplyr, which we'll be using for our next demo, we would just type install. packages and specify the name dplyr. When we press Enter, R will go online to the CRAN Repository, which stores all of the public R packages and download and install dplyr on our machine. Once a package has been installed, to use it in our code, simply type library and specify the package we want to load in memory. Now we have access to all of the commands and features found in the dplyr package. If you need help with any of the commands in R, just type question mark and the name of the command that you need help with. This will pop up the online documentation for that command in the help pane in the lower right-hand corner. From there, you can learn all about the commands, their arguments, return values, and more. Other than what we've seen in this demo so far, R contains most language features you'd expect from other C-like programming languages. For example, control structures, like if/then/else statements, for loops, while loops, and switches, mathematical, relational, and logical operators, and string manipulation functions with regular expressions. The R language has evolved quite a bit over the years and there are many more language features available than we have time for in this brief overview.

Summary

In this module, first, we learned about the R programming language and why it has become so popular in recent years. We also learned about RStudio, the IDE that we'll be using for the remaining demos in this course. Then, we learned about the basic syntax and operations for the R programming language. We learned about the data types in R, data structures, data frame, and indexing and subsetting data. In the next module, we'll learn how to work with data using R.

Working with Data

Introduction

Hello again, and welcome to our next module on Data Science with R. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to work with data using R. As an overview of this module, first, we'll learn how to work with data using R. That is, we'll learn how to import, clean, transform, and export data. We'll also learn how R makes this process easier and more efficient than by using other tools. Then, we'll see a demonstration where we'll import, transform, and export data. The data transformation steps will be done using a very popular extension package in R called dplyr. So let's get started.

Working with Data

When we're working with data prior to performing a data analysis, there're typically four main steps we perform in R. These four steps are often necessary to transform our raw data into a form suitable for analysis. First, we need to import our raw data from one or more data sources into R. Next, we need to inspect and then clean our raw data so that we can work with it effectively and produce reliable results. Then we need to transform our clean data into a form suitable for the analysis we intend to perform. Finally, we often need to export our data into a different format so that it can be shared with others. You'll often hear these steps referred to as data munging, data wrangling, data cleaning, data cleansing, or ETL, that is, Extract, Transform, and Load, depending upon the context and the specific user community. All four steps may not always be necessary, but in general we'll typically use some combination of these four steps. Next, we'll take a look at each of these four steps and how R makes our job easier during each of these steps. The first step when working with data in R is to import our data into R. Luckily for us, R makes it very easy to import data from a wide variety of data sources. We can load data from file-based data sources, such as comma separated values, tab delimited values, and Excel files, web-based data sources, for example, XML, HTML and JSON data, databases, for example SQL Server, Oracle, and MySQL, and statistical data files like SAS, SPSS, and Stata files, and there's many more. In fact, there are literally hundreds of data sources that R supports using downloadable extension packages. In addition, R is very flexible in the way that it can load data, which allows it to handle a wide variety of data scenarios. The second step when we work with data in R is to clean our raw data so that we can work with them effectively and ensure that they'll provide reliable results. This step is often necessary, because most raw data are not typically stored in a format suitable for analysis.

In addition, many raw-data sources contain errors and anomalies that need to be corrected in order for our analysis to produce reliable results. There are many tasks typically involved in the process of cleaning data. Some of these tasks include reshaping data so that we have a table of data with each variable stored as a column and one observation per row; renaming columns so that we have accurate and human readable column names; converting data types from one data type to another, that is, converting between logical value, integers, numerics, character strings, dates, times, and factors; ensuring that all data types are properly encoded, for example converting the letters m and f to the categories male and female, respectively; ensuring all data are internally consistent by converting to the same unit of measure, the same numeric scale, and using the same internal format for numbers, dates, and times; handling both errors in our data, and correcting outliers, for example, values that were incorrectly typed into a computer by hand, or were the result of a sensor misreading, and handling missing values, either by omitting the entire row, or imputing a suitable replacement value. R has a series of built-in tools and extension packages that make cleaning our data significantly easier than by using alternative methods or by cleaning the data by hand. We have a wide variety of built-in functions in the base packages in R; that is, the packages that are part of the initial R installation. These functions include functionality for handling many of the data cleaning tasks that we just discussed and more. In addition, we have several extension packages, that is, packages that are not part of the base installation that must be downloaded and installed, designed to help with more specific data-cleaning tasks. For example, when we have extension packages like tidyr, which is focused specifically on tidying data; that is, reshaping data from a variety of non-standard formats to a standard tabular data format referred to a tidy data; that is, we have all variables on the columns and one row for each observation. Reshape2, which offers the same functionality as tidyr, but provides more functionality for reshaping and aggregating data at the expense of a more complex syntax than tidyr. Stringr, which provides a simple and consistent set of commands for character-string manipulation, lubridate, which provides a simple and consistent set of commands to work with dates, and validate, which allows you to create a set of reusable domain rules to check if your data adhere to those expected rules. The third step when we're working with data is typically to transform the data into a form suitable for analysis. This typically involved several steps, including selecting the columns that we want to include in our analysis, filtering the rows that we want to include in our analysis, grouping data together based on one or more columns and aggregating the results within each group, ordering the rows of our data by one or more columns, which we refer to as sorting or arranging, and merging one or more tables of data together. There are several other tasks that we might perform during the data transformation process. However, these are the most common tasks. In addition, there's a lot of overlap between the data cleaning

and data transformation steps. In fact, many people simply group these into a single task. What's most important is that you end up with data that is both clean and transformed into a format suitable for analysis. There are several built-in functions and extension packages available in R to help us transform our data. For example, the base R installation contains functions for selecting columns, filtering rows, grouping rows, and sorting rows as well. We saw a few of these functions during a previous demo on subsetting data frames. Plyr provides tools for splitting, applying, and combining data within R. Dplyr is the next iteration of plyr designed to provide a consistent grammar for data manipulation using data frames, data tables, and databases. It has several improvements over its predecessor, plyr. Data. table provides similar functionality to both plyr and dplyr, but has a syntax similar to the subsetting syntax we saw in our previous demo on subsetting data frames. And the sqldf package allows us to execute queries using SQL, that is Structured Query Language, on data contained within our data frames. It's useful to know all five of these methods for transforming data in R. However, we'll be focusing specifically on dplyr during our upcoming demo, since it's both very powerful, and yet easy to understand. The fourth and final step when working with our data is often to export our cleaned and transformed data into a new data repository. We don't always have to perform this step. For example, if we move right into analyzing the data after we've imported, cleaned, and transformed the data, however, many times we want to share cleaned and transformed data so that others can analyze these data. Once again, luckily for us, R makes exporting data extremely easy by providing us with support for a wide variety of data repositories. We, again, have support for file-based data formats, like CSV, tab, and Excel files, web-based formats, like HTML, XML, and JSON, databases, for example SQL Server, Oracle, and MySQL, statistical data files, for example, SAS, SPSS, and Stata files, and many more. Before we transition into our demo, let's take a quick look at dplyr and what it can do to help us with our data transformation process. Dplyr is a popular extension package for R that is designed to provide a consistent grammar for data manipulation. It was designed to work with data frames, data tables, and with supported databases. It's very similar to SQL, that is Structured Query Language, in terms of both its function and purpose. Dplyr is composed of a set of verbs that allows to manipulate data within a single table or across multiple tables. The single-table verbs in dplyr include, but are not limited to, select, which selects a subset of columns from our table, that is, it creates a projection; filter, which selects a subset of rows from our table, that is, it filters the data based on a set of criteria; mutate, which creates a new column by transforming data in existing columns into a new form; summarize, which allows us to summarize or aggregate a set of rows that are grouped together using a common value; and arrange, which reorders the rows in a table, based on one or more columns in an order within which to sort them. The multi-table verbs, which operate on two or more tables include

operations you'd expect, like joins, union, intersection, and set difference. However, we're just going to focus on the single-table verbs during our upcoming demo. Unfortunately, preparing your data for analysis is often the most difficult and most time-consuming step in the data science process. In fact, most data scientists, data analysists, and consultants, myself included, report spending about 80% of our time, on average, on this step alone. The unfortunate reality is that while data science seems like a glamourous job, most data scientists spend most of their time doing data janitor work just to get their data prepared for analysis. So my recommendation to everyone is to record all steps using a script so that you can reapply the steps whenever they are needed, because, it is inevitable that you'll be on step 50 of a complex data analysis and realize that you've made a mistake back in step 5. You don't want to have to go back and reapply every step by hand, you just want to make the change at step 5 and re-run the script instead. In addition, once you've performed a data analysis, you'll likely be asked to perform it again a year later with the latest data. If you have your data transformation steps recorded as a script, you can just point it at the new data, and re-run the script. So now, let's see how to transform our new data in R using dplyr.

Demo

For the next few demos in this course, we'll be using a very popular data set in R called the Motor Trend Cars Data Set. This data set was collected from cars contained in Motor Trend magazine in 1974. These data contain a series of measures from 32 cars released between 1973 and 1974, however, for our demo we're just going to be using a subset of these measures to keep things simple. Our data set will contain four columns, which are referred to as variables in this context. Name, which is the make and model of the car; Transmission type, which is a categorical variable containing manual for a manual transmission, and automatic for an automatic transmission; Cylinders, which is an integer representing the number of cylinders contained in the car's engine, for example, a V-6 engine would be represented as the integer 6; and, Fuel Economy, which is a numeric data type, which represents the rate at which the car consumes fuel, measured in U. S. miles per gallon of fuel. The data set contains 32 rows, 1 row for each car contained in the data set. During this demo, first, we're going to import our data from a tab-delimited flat file. Next, we'll transform our data using dplyr. Finally, we'll export our data to a comma-separated values file that is a CSV file. Cleaning data can often be a long and complex process, so I've chosen to omit this step for our demo to keep things moving along and interesting. I just want to show you the most important aspects of using R right now, so that you can choose to invest time learning how to clean data from a more in-depth course later if you choose to do so. I'll point you toward

these more in-depth courses for cleaning and transforming data in the last module of this course. First, let's take a look at the tab-delimited flat file that we're going to be using as our data source. As we can see, we have a text file document containing a header, that is, the names of the columns are contained in the first line of text in the file. We have four columns of data in our data set, that is, Name, Transmission, Cylinders, and Fuel. Economy. Each column is separated, that is, delimited by the tab character, which is represented in the text editor as an orange arrow. The end of each row of data is delimited by the carriage return line feed character sequence, which is represented by the CR and LF symbols. The file contains 32 lines of text for each of the 32 rows of data, plus 1 extra line for the header. In addition, text data is denoted by using the double-quote character on both sides of the textual data. This type of file is pretty standard for exchanging data in the IT industry. So now, let's switch over to R to begin working with these data. To begin working with file-based data, first, we need to set our working directory to the directory on our computer that contains our data files. We're going to be using a directory called Pluralsight, located at the root of the C: drive of our computer for all the demos in this course. We set the working directory using the setwd function, and we pass in the path to our directory into the function as an argument. For example, in our case, we'll pass in C:/Pluralsight as our working directory. Please note that in R, we typically use the forward slash character as our path separator character even when we're using a Windows-based system. This is because the backslash character is used to indicate a character escape sequence, so using the forward slash is simpler and easier to read. Next, we need to import the data into R. As we mentioned earlier, there are a wide variety of ways to load data into R. We're going to be loading a tab-delimited data file, so we use the read. table function, set the file name parameter to Cars. txt, which is the name of the tab-delimited data file, set our header parameter to TRUE, indicating that our first row contains the column names for our data set, set the separator parameter to the tab character, which is represented by the character escape sequence \t, this tells R that the columns of data are separated by the tab character, and set the quote character, that is the text delimiter, to the double-quote character using the character escape sequence \"". This tells R that anything contained inside of a pair of double-quote characters should be interpreted as text, including special characters, like the tab character. We'll then assign the data that we've loaded into a data frame called cars. Now, let's take a quick peak at these data that we've loaded. We'll do this using the head function, which displays just the first six rows of data. As we can see, we have four columns, that is Name, Transmission, Cylinders, and Fuel. Economy. In addition, we can see the first six rows of our data. To begin transforming our data, we need to load the dplyr package. We already downloaded and installed this package during our previous demo, so we don't need to do that again. In addition, going forward in these demos, I'm going to skip the download and install

steps, and move right into loading packages into memory. This will help keep things moving along more quickly. We do, however, still need to load the dplyr package into memory. We'll do this using the library function and pass in dplyr as an argument. Now, we have access to all of the functions and features of the dplyr package. We're going to transform our data step by step, performing one transformation step and then inspecting the results of each step to see the results of our transformation. To transform our data, we're going to select just three columns that we want to work with. That is, Transmission, Cylinders and Fuel. Economy. We'll do this using the Select function in dplyr, set our data parameter to our cars data frame, and please note that with dplyr our data parameter is actually. data, rather than just data, which was done intentionally to avoid name collision. Then we'll specify the three columns that we want to keep from the four columns available in our data frame. We separate each column name using a comma, and we'll assign the results of this transformation to a variable called temp, for temporary. Now, let's see what the results of this transformation look like using the head function on our temporary variable. As we can see, our results contain only the three columns we specified, that is, Transmission, Cylinders, and Fuel. Economy. Next, we'll filter our data to include only cars with automatic transmissions. We'll do this using the filter function, set our data parameter to our new temporary data frame, and we'll add a filter predicate specifying to include records where Transmission is equal to Automatic. And please note that we could have used one of many types of predicate functions including inequality operators and more complex matching expressions. Now let's take a look at the results of our filter transformation. As we can see, we now have only rows where Transmission type is set to Automatic. Next, we'll convert our Fuel. Economy values from U. S. miles per gallon of fuel to the metric equivalent fuel consumption in kilometers per liter. We'll do this using the mutate function, set our data parameter to our temporary data frame variable, set our mutation expression to Consumption set equal to Fuel. Economy times 0. 425. This tells dplyr that Consumption will be the name of the new column we are creating using the existing Fuel. Economy column and multiplying each value by 0. 425, which is the conversion factor from miles per gallon to kilometers per liter. In addition, please note that the phrase Fuel Consumption is the metric equivalent name for what is referred to as Fuel Economy in the U. S., hence the new column name. Now, let's inspect the results of this transformation. As we can see, we now have a fourth column called Consumption that contains the metric equivalent of each fuel economy value. Next, we'll group our rows of data by the number of engine cylinders that each car contains. We'll do this using the group_by function, set the data to our temp variable, and set group_by to our Cylinders variable. Now let's inspect the results. As we can see, our results have changed from our familiar format to something that looks slightly different. This is because the group_by function returns a grouped data frame rather than a regular data frame as its result.

The printed output of our group data frame tells us that the data source is a data frame with six rows of data being displayed, which is the default for the head function and four columns of data. It also tells us that the number of groups in the result is 3, that is, we have 4, 6, and 8 cylinder engines. In addition, it also tells us the data type of each column under the column name, that is, fctr for Transmission, int for Cylinders, dbl, which is an alternative name for a numeric variable for Fuel. Economy, and dbl for Consumption. Now that we have our data grouped, we can summarize the values contained in each group. We do this using the summarize function, set our data parameter to our temp variable, and set our summarization expression to Avg. Consumption set equal to the arithmetic mean of Consumption. This tells dflyr to create a column in our results called Avg. Consumption, which will contain the average consumption for each group of cars by the number of engine cylinders they contain. Now let's take a look at the results of our transformation. As we can see, we have a new data structure called a tibble, which contains 3 rows and 2 columns. A tibble is a modern take on a data frame that contains all of the best features of a data frame, but eliminates some unnecessary features. This tibble contains two columns, that is, Cylinders, which is an integer, and Avg. Consumption, which is a double, that is a numeric value. The three rows of which there is one for each number of cylinders, contain the average fuel consumption in liters per kilometer for all cars in each group, based on the number of cylinders they possess. Next, we'll arrange the order of the rows so that we sort our rows by average fuel consumption in descending order. That is, we want the most fuel-efficient vehicles on the top, and the least efficient on the bottom. We'll do this using the arrange function, set our data parameter to our temporary variable, and set our sort expression to Avg. Consumption in descending order. Now let's take a look at the results of this transformation. As we can see, we have our tibble of data sorted by Avg. Consumption in descending order. However, since our data was already in this order merely by coincidence, we didn't really see any changes from our previous results. Finally, we'll convert our tibble back into a data frame. We'll do this using the as. data. frame function, passing in our temporary variable as an argument. Then, we'll assign the result to a new variable called efficiency. Now, let's inspect the results of this final transformation. As we can see, we've converted our transform data contained in the tibble back into a data frame. So far in this demo, I've shown you how to use dplyr in a step-by-step manner, calling out each step and each parameter involved in each step. While this made the initial demonstration more clear and straightforward, it is, however, both inefficient and not as easy to read when creating production code. Dplyr is a grammar for data manipulation, and as such, we can write our data manipulation code in a much more succinct way by using what is referred to as fluent notation using piping, also known as method chaining. The pipe operator in dplyr is the percent, right angle brace, percent character sequence. We can pipe each of our dplyr functions together line

by line to create a much more readable chain of commands. To demonstrate how this works, we'll execute the exact same set of commands that we used previously, however, this time we'll chain the commands together using the pipe operator instead. We'll start by assigning the results of our data manipulation to a variable called efficiency, set our data source to our cars data frame, select the three columns, filter to include only automatic transmissions, mutate to create our metric fuel consumption variable, group by Cylinders, summarize by the average consumption of each of our cylinder groups, arrange our rows by average consumption in descending order, and convert our results to a data frame. Now let's inspect the results of this chain set of commands. As we can see, the Chain version of our command produces the exact same results as our step-by-step commands. Finally, we'll export the results of our data transformation steps to a comma-separated values file. We'll do this using the write. csv function, set our x parameter to our efficiency data frame. Please note that the parameter x is just a generic parameter name for the data we want to save, set our file name to Fuel Efficiency. csv, and set our row. names parameter to FALSE, indicating that we do not want to write the unique id for each row to our CSV file. Now let's open us this CSV file to see the results. As we can see, we now have a comma-separated values file containing the results of our data transformation. The file contains a header row specifying the names of our two columns, that is, Cylinders and Avg. Consumption. Each field of data is separated using a comma, and each of our rows is delimited using the carriage return linefeed character sequence. This is just a small taste of the power that R provides to import, clean, transform, and export data. There's so much more that I'd like to show you, however, in order to keep things moving along, we'll have to defer this additional information to a more in-depth course on working with data in R. I'll point you in the direction of these courses during the end of the last module of this course.

Summary

In this module, first, we learned how to work with data in R. We covered importing, cleaning, transforming, and exporting data. Then, we saw a demo where we imported, transformed, and exported a data set. In addition, we learned how to use dplyr for our data transformation steps. In the next module, we'll learn how to create descriptive statistics using R.

Creating Descriptive Statistics

Introduction

Hi there, and welcome back to Data Science with R. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to create descriptive statistics using R. As an overview of this module, first, we'll learn about descriptive statistics and how we can use R to describe our data in numerical ways. We'll cover the standard statistical measures for both a single categorical variable and a single numeric variable. Then, we'll see a demo where we'll learn how to create these descriptive statistics using R. We'll create frequency tables, measures of central tendency, measures of spread, and we'll create correlation coefficients as well. So let's get started.

Creating Descriptive Statistics

Descriptive statistics describe data in numerical ways. In statistics, they're the foundation of the numerical analysis of data. Descriptive statistics are how we typically summarize the shape and feel of our data in numerical ways. Hence, they're often referred to as summary statistics. For example, if we want to summarize the values contained in our fuel economy variable, we could create a set of statistics that describe these data like the descriptive statistics on the right. Simply by inspecting these statistical measures, we can learn quite a bit from the distribution of values contained in this variable. I don't want to go too deep into statistics in this course, since I'm assuming that you've already had some exposure to basic statistics before watching this course. However, just to make sure we're all on the same page with our terminology, let's review a few basic terms from statistics. First, we have observations, which are essentially the rows in the table. They are referred to as observations, because in statistics we're typically interested in observations of some kind of physical phenomenon, like the average temperature each day. Next, we have variables, which are essentially the columns in the table. They're called variables because their values vary across each observation or row in the data set. In addition, variables can generally be subdivided into categorical variables, which represent qualitative values, like a car's name and transmission type, and numeric variables, which are quantitative values, like the number of cylinders in fuel economy and miles per gallon. We'll be using these terms throughout this course to help us conceptually organize the types of data analysis that we'll be performing. Categorical variables contain qualitative values, that is, named values like the type of transmission in our cars data set. When we're creating descriptive statistics for our categorical variable, we're typically interested in the frequency, or count of observations of each category in our categorical variable. We generally represent the frequency of observations for each category in our categorical variable using a frequency table, like the frequency table on the right. In the first column in this table, we have our categorical variable, that is Transmission type, which contains

two categories, that is Automatic and Manual. In the second column, we have the Frequency of observations of each category, that is, the total number of cars in our data set containing the respective transmission type. From this table, we can see that our data set contains 19 cars with Automatic transmissions and 13 cars with Manual transmissions. Numeric variables contain quantitative values, like the fuel economy of our cars in U. S. miles per gallon. When we're creating descriptive statistics for a single numeric variable, we're typically interested in describing where the center of the distribution of values is located. We refer to these as measures of central tendency. For example, we can see that the arithmetic mean of our Fuel Economy is 20. 09 miles per gallon. In addition to measures of centrality, we're also typically interested in measures of the spread of the data, also known as dispersion in statistics. For example, we can see that the standard deviation of our distribution of values is 6. 02 miles per gallon. There's one more type of descriptive statistic that I'd like to discuss before moving on to our next demo. Correlation is the measure of the strength of the relationship between two numeric variables. It's essentially the degree to which two variables vary with one another. A correlation coefficient provides us with a measure of the strength of this relationship on a standardized scale. This standardized scale ranges from -1, that is a perfect negative correlation, to +1, that is a perfect positive correlation, with 0 indicating no correlation at all, that is, the 2 variables are completely unrelated. For example, in the table on the right we can see that there is a negative that is inverse correlation between the number of cylinders in the engine of a car and the fuel economy. This tells us that fuel economy decreases as the number of cylinders increases. In addition, this value is -0. 85, which tells us that this is a relatively strong correlation. If we look back at the fuel economy table we created in our last demo, this should match our intuition; as the number of cylinders increases, the average fuel consumption in kilometers per liter decreases. There are many other descriptive statistics that we can create for both categorical and numeric variables and for various combinations of these types of variables. However, just to keep things moving along, we're just going to demonstrate how to create these most common descriptive statistics. Creating the other types of creative statistics works in roughly the same way. So now, let's learn how to create these descriptive statistics for our categorical and numeric variables.

Demo

Now let's see how to create descriptive statistics using R. For this demo, we're going to attempt to answer 3 questions about our data set of Motor Trend Cars from 1974. We want to know how many cars we have, based on their type of transmission, that is, automatic or manual transmission. We'll answer this question by creating a frequency table of our categorical

transmission variable. The second question that we'll answer is what does the distribution of fuel economy values look like? We'll look at a variety of numerical descriptive statistics, like mean and standard deviation, to answer this question. Finally, how are the number of engine cylinders and the fuel economy related? We'll answer this question by creating a correlation coefficient of these two variables. To begin creating our descriptive statistics, first we need to load our cars data set into R. Rather than loading the tab-delimited file we used in our previous demo, we're going to load a comma-separated values file containing the exact same data instead. First, we'll set our working directory to the directory containing our data file. Next, we'll load our data into a data frame called cars using the read. csv function, passing in our file name, that is Cars. csv, as an argument. We'll just leave all of the other parameters like header, separater, and quote character to their defaults. Now, let's take a quick peek at the first six rows of these data to see what we've loaded. As we can see, we have the same data as we had previously loaded. The first question we want to answer is how many cars to we have of each transmission type? Transmission is a categorical variable, so we'll create a frequency table to answer this question. To create a frequency table, we'll use the table function and pass in a reference to the Transmission variable contained in our cars data frame. As we can see, this creates a frequency table of our Transmission variable indicating the frequency of observations for both cars with Automatic transmissions and cars with Manual transmissions. The second question we were attempting to answer is, what does the distribution of values in our Fuel. Economy variable look like? Fuel. Economy is a numeric variable, so we'll answer this question by creating a series of descriptive statistics for this variable. First, let's get the minimum value. We'll do this using the min function and pass in a reference to our Fuel. Economy variable in our cars data frame. As we can see, the car with the lowest fuel economy is 10. 4 miles per gallon. Next, we'll get the maximum value using the max function. As we can see, the car with the highest fuel economy is 33. 9 miles per gallon. Then, we'll get the average value using the mean function. The average fuel economy for all vehicles in our data set is just over 20 miles per gallon. We'll use the median function to get the median value, which is 19. 2 miles per gallon. To get the quartiles, we use the quantile function. This gives us the minimum, first quartile, median, third quartile, and maximum value. Next, we'll get the standard deviation of the distribution of values for our fuel economy variable. As we can see, the standard deviation is just over 6 miles per gallon. Finally, we can get the total of all fuel economy values using the sum function. The total of all fuel economy values is 642. 9 miles per gallon. However, summing the average fuel economy of each vehicle probably wouldn't provide us with any meaningful information in the real world since we're taking the sum of the averages. Our third question is how does Cylinders and Fuel Economy relate to one another? Both Cylinders and Fuel. Economy are numeric variables, so we can create a correlation coefficient to

see how they're related to one another. To do so, we use the correlation function, set the x parameter to our Cylinders variable, and set the y parameter to our Fuel. Economy variable. As we can see, the correlation coefficient for these 2 variables is -0. 85, which is a relatively strong inverse correlation. In the event you'd like to summarize an entire table of values, we can use the summarize function and pass in a data frame as an argument. When we summarize a table, it provides us with a summary of each variable, based on the type of variable in each column. For categorical variables, like Transmission, it returns a frequency table for the variable. For numeric variables, it returns the five-number summary statistics, plus the mean. Five-number summary statistics include the Minimum, First Quartile, Median, Third Quartile, and Maximum. R has the most extensive collection of commands for creating descriptive statistics that I've seen so far in a programming language. There are so many more statistical concepts that we could look at that we could fill an entire course on this topic alone. However, this should cover the most important summary statistics you'll be using on a day-to-day basis, and it'll point you in the direction of where to go to learn more about descriptive statistics with R at the end of the last module of this course.

Summary

In this module, first, we learned about descriptive statistics and how they can be used to describe our data in numerical ways. We covered descriptive statistics for both categorical and numeric variables. Then we saw a demo where we learned how to create these descriptive statistics using R. We also learned how to summarize an entire table of data as well. In the next module, we'll learn how to create data visualizations with R.

Creating Data Visualizations

Introduction

Hello again, and welcome back to Data Science with R. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to create data visualizations using R. As an overview of this module, first, we'll learn about data visualization and the three main plotting systems in R. Then we'll look at a series of standard data visualizations and explore some more advanced data visualizations as well. Then, we'll see a demo where we'll learn how to create a few of these data visualizations using R. So let's get started.

Creating Data Visualizations

Data visualization is the representation of data by a visual means. We do this because the human brain is exceptionally good at visual pattern recognition, so the idea is to map the properties of the data to visual characteristics, like location, size, shape, and color. R has very powerful tools to help you quickly create a variety of data visualizations for your data. There are three main plotting systems in R. The base graphic system, lattice, and ggplot2. We'll take a closer look at each of these plotting systems next. The first plotting system is called the base graphic system. This is the graphic system that is built into R and provides basic data visualization capabilities. This is an extremely flexible system that allows you to build your plots through a series of high-level plotting functions, mixed with low-level graphics functions as well. For example, we can create a chart using the high-level plot function and then make modifications to the defaults by specifying additional parameters or by calling low-level graphics functions. The second plotting system is called lattice. It's based on trellis graphics, which was originally developed for the statistical programming languages S and S+, which were two predecessors of R. It's designed to make it very easy to create multi-varied data visualizations. That is, data visualizations involving many variables. In addition, its formula based, so we define our data visualization in terms of a formula. It works very well for basic plotting, but it's also flexible enough to handle most non-standard needs as well. Lattice is available as a free, open source, downloadable, extension package. The third plotting system is called ggplot2. Ggplot2 is based on The Grammar of Graphics, a thesis on statistical graphics written by Leland Wilkinson in 1999. It attempts to simplify the creation of data visualizations by using a high-level language to represent the various aspects of data visualizations. With ggplot2, we work with higher-level extractions like layers, geoms, aesthetics, scales, and more. In addition, we can build up our data visualizations by composing or chaining various functions together. Ggplot2 is also available as a free, downloadable extension package for R. Each of these three plotting systems has various pros and cons. In addition, it might be difficult or impossible to create a specific data visualization in one or more of these plotting systems. So, I recommend you learn about each of them and their unique strengths and weaknesses. However, to keep things moving along, we're just going to look at ggplot2 during our upcoming demo since it's currently the most popular of these three plotting systems. With the three plotting systems in R, it's possible to create just about any standard data visualization. We can create frequency bar charts to visualize a single categorical variable. We can create data visualizations like histograms and density plots to visualize a single numeric variable, scatter plots to visualize two numeric variables, and more advanced data visualizations for 2 variables, like 3D density plots. R includes a wide variety of standard data visualizations for all combinations of one

or two categorical or numeric variables. In addition, we can create faceted data visualizations to visualize three or more variables at the same time, or we can create a scatter-plot matrix to view the relationship between many variables all at once. Beyond this, we can create data visualizations for other types of data, like spatial data, that is data that are located on the surface of the earth; hierarchical data, that is, data organized in a tree-like structure, network data, that is, the relationships between entities and our data set, and textural data, that is, data that are contained in a body of text. Finally, we can create animated data visualizations and interactive data visualizations as well. As you can see, R has very powerful and flexible data visualization capabilities. In fact, there is so much you can do with R that it would take multiple courses just to cover all of these topics. However, for our upcoming demo, we're just going to keep things simple and stick to a few standard data visualizations. Then, I'll point you towards additional courses on data visualization with R at the end of the last module of this course.

Demo

For a data visualization demo, we're going to attempt to answer three questions about our data set of cars. These are the same questions we answered in the previous demo, however, we'll answer these questions using data visualizations instead of descriptive statistics. Once again we want to know how many cars we have based on the type of transmission, that is, whether we have an automatic of manual transmission. We'll answer this question by creating a frequency bar graph of our categorical transmission variable. Next, what does the distribution of fuel economy values look like? We'll create a histogram and a density plot to answer this question. Finally, how does the number of engine cylinders and the fuel economy of a car relate to one another? We'll answer this question by creating a scatter plot of these two variables. We'll begin our data visualization demo again by loading our cars data into memory. First, we'll set our working directory, then we'll read in our CSV data into a data frame called Cars. Next, we'll create our data visualizations using ggplot2. We need to download, install, and load ggplot2 into memory first. I've already downloaded and installed ggplot2 in the same way we saw at the end of the first demo, so we're just going to load it into memory now. The first question we're attempting to answer in this demo is how many cars we have of each transmission type. The Transmission variable is a categorical variable, so we'll create a frequency bar chart to answer this question. A frequency bar chart is the visual equivalent of a frequency table. To create a frequency bar chart using ggplot2, we use the ggplot function, set the data parameter to our cars data frame, set the aesthetics of the chart with the X axis set to the transmission variable contained in our cars data frame. The aesthetics tell ggplot2 how to map the properties of the data to visual characteristics

like location, size, and color. Then we add a bar geom. A geom, which is short for geometry, tells ggplot2 what type of chart to create. Using a bar geom tells ggplot2 to create a bar chart. Then, we add a main chart title, add an X access label, and add a Y axis label. As we can see, this creates a frequency bar chart with the Transmission Type on the X axis, and the frequency of observations for each type of transmission on the Y axis. The second question we're attempting to answer is what does the distribution of Fuel. Economy look like? Fuel. Economy is a numeric variable, so either a histogram or a density plot would be a good way to answer this question. To create a histogram, we use the ggplot function, set the data parameter to our cars data frame, set our aesthetics with x set to our Fuel. Economy variable, add a histogram geom to tell ggplot2 to create a histogram, set the number of bins to 10, indicating that we want to create equidistant bins to group our data into, add a main chart title, add an X axis label, and add a Y axis label. As we can see, this creates a histogram with Fuel. Economy in miles per gallon on the X axis and the frequency of occurrences contained within each of the 10 bins in the histogram on the Y axis. We could also change the number of bins to create either a more coarse-grained or more fine-grained histogram as well. Let's also create a density plot to help us better visualize the shape of the distribution of Fuel. Economy values. To create a density plot in ggplot2, we use the same function and parameters as our histogram, however, this time instead of adding a histogram geom, we'll add a density geom instead. As we can see, this creates a density plot for the distribution of our Fuel. Economy variable. On the X axis we have Fuel Economy in miles per gallon, and on the Y axis we have the probability density, which represents the likelihood of an observation occurring at a specific fuel economy value. Our third question is how do the number of cylinders in the Fuel. Economy relate to one another? Both Cylinders and Fuel. Economy are numeric variables, so we can create a scatterplot to visualize their relationship. To create a scatterplot, we use the ggplot function, set the data parameter to our cars data frame, set the aesthetics with x set to Cylinders and y set to Fuel Economy, add a point geom to tell ggplot2 to create a scatter plot, add a main chart title, add an X axis label, and add a Y axis label. As we can see, this creates a scatterplot with the Number of Cylinders on the X axis and the Fuel Economy on the Y axis. Each point represents an observation of a car located at the specific X/Y coordinates based on the number of Cylinders and the corresponding Fuel Economy. In addition, if we were to draw a line of best fit through all these points, we'd have a line with a downward slope. This tells us that the number of cylinders and the Fuel Economy are inversely related to one another. That is, as the number of Cylinders goes up, the Fuel Economy goes down. This should match your intuition, based on the fact that the correlation coefficient we created in the previous demo was a negative value. We're really just scratching the surface of data visualization with R in this course. There are many more types of data visualizations that can be created in both the

three plotting systems we discussed, and a variety of other more specialized data visualization packages as well. However, we have plenty more topics to cover in this course. So, we'll have to wait until the end of the final module to find out where to go to learn more about data visualization with R.

Summary

In this module, first we learned about data visualization and the three main plotting systems in R. We looked at a few standard data visualizations and some more advanced data visualizations as well. Then we saw a demo where we learned how to create a few of these data visualizations using R and ggplot2. We learned how to create frequency bar charts, histograms, density plots, and scatterplots. In the next module, we'll learn how to create statistical models using R.

Creating Statistical Models

Introduction

Hello again, and welcome to the next module in our introduction to Data Science with R. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to create statistical models using R. As an overview of this module, first, we'll learn about statistical modeling and how we can use statistical models to create inferences and make predictions. In addition, we'll learn about simple linear regression models. Then, we'll see a demo where we'll learn how to create a simple linear regression model using R.

Creating Statistical Models

A statistical model is a set of one or more mathematical equations used to represent a sample of data. That is, it's a mathematical model based on probabilities that attempts to describe the process that created these data. A model is a simplification or an approximation of reality, that is, it's a mathematical abstraction that captures the essence of reality, but does not perfectly describe it. In the most general sense, a model includes any abstract representation of our data. This would include a probability density function and even a data visualization. However, in statistics, the term statistical model is generally used to mean a model of the relationship between two or more variables. Statistical models serve several purposes. A few of the main

purposes include description, that is, we can use a statistical model to describe the basic characteristics of a sample of data that we're observing. Inference, we can use a statistical model of a sample of data to estimate the parameters of a larger population. Comparison, we can compare sets of data to determine if they're different in a way that is statistically significant, for example, hypothesis testing. And prediction, we can use statistical models to make predictions about new, unknown observations. There are two main categories of statistical models. There are parametric models, where we have a fixed number of parameters and our model is completely defined by setting those parameters. For example, the Gaussian distribution, which is the bell-shaped curve on the right, is completely described by two parameters. That is, the mean and the standard deviation. Simply by knowing the values of these two parameters, we can model the source that generated these data and predict where future unknown values are likely to occur. In addition, there's also non-parametric models, where the number and nature of parameters are not fixed in advance, but grows with the amount of data available to create the model. Non-parametric models are a bit more complicated than parametric models, so to keep things simple, we're just going to focus on parametric models in this course. There are numerous types of statistical models that we can create in R. So many, in fact, that we couldn't possibly cover them all in a single module, let alone an entire course. Within R, we can create a wide variety of statistical models; these models include simple, single-variable models like a probability density function, which is a parametric model we use to represent various statistical distributions of values. In addition, we have models that involve the relationship between two or more variables, like analysis of variance, or a nova, which allows us to compare the difference of the means among three or more groups of data, simple linear regression, which allows us to model the relationship between two variables as a linear function, and more advanced models like non-linear regression, which allows us to model the relationship between two variables when their relationship does not form a straight line, and Bayesian networks, which are very powerful statistical models, which model the relationship between many variables as a network of conditional probabilities based on Bayesian statistics. To keep things simple, however, we're going to focus on only one relatively simple, yet highly valuable type of statistical model, a simple linear regression model. A simple linear regression model is a type of statistical model used to model the relationship between two variables. That is, how change in one variable influences a change in a second variable. It's referred to as a linear model, because it attempts to model this relationship using the mathematical equation for a straight line. Simple linear regression involves two variables. First, we have an explanatory variable, which is also referred to as an independent variable, or a predictor variable, depending upon the context. We typically plot this variable on the X axis of a chart. In simple linear regression, this variable is a continuous numeric variable.

Second, we have a single scalar outcome variable, which is referred to as a dependent variable or response variable, depending upon the context. We typically plot this variable on the Y axis of a chart. In simple linear regression, this variable is also a continuous numeric variable. With simple linear regression, we're using a linear predictor function, which is a fancy way to say that we're using the equation for a straight line. The equation for a straight line, as you may remember from high school math, is y = m * x + b. In this equation, y is our outcome variable and x is our explanatory variable. M is the slope of the line, that is, for each unit increase in x, how much does y increase, and b is the y intercept, that is, the y value in x is equal to 0. We have two parameters that need to be set in order to create this model, that is, m, the slope, and b, the y intercept. We use an algorithm known as the method of least squares to provide estimates for these parameters based on the data. That is, we use the data to determine the slope, m, and the y intercept, b, of a line that best fits all of these data points. The resulting line minimizes the squared vertical distance between each point and the regression line. This means that the resulting line fits the data better than any alternative line we could draw through our set of data points. Linear regression makes a few key assumptions about the data, which include linearity, that is, the source that generated the data follows a straight line, and homoscedasticity, that is the very ability of the data stays roughly the same over the length of the line. If these assumptions are incorrect, simple linear regression is not valid for these data. Just to demonstrate how violating these assumptions can create invalid linear regression models, let's take a look at four examples of simple linear regression. In the first plot, we have an example of a good linear regression model, that is, the linear regression line is a good fit for these data. However, in the second plot, we have data that violate the linearity assumption, that is, the data are non-linear, and thus our linear regression line is not a good fit for these data. In the third plot, we have an outlier throwing off our linear regression line. And in the fourth plot, we have an outlier creating the appearance of a high correlation, even though the data are linearly correlated. This is why it's important that we understand the assumptions a statistical model is operating under and that we take the time to explore our data for problems like these before creating our statistical models. Now, let's learn how to create a simple linear regression model using R.

Demo

For our next demo, we're going to use a new data set. This data set is called the Iris Data Set. It contains 150 observations of 4 measurements of 3 species of Iris flower. The three species are Iris Setosa, Iris Versicolor, and Iris Virginica. The four measurements that this data set contains are petal length, that is, the average length of the petals of the flower in centimeters, petal width, that

is, the average width of the petals of the flowers in centimeters, sepal length, that is, the average length of the sepals, which are the leaf-like structures that enclose the petals until the flower blooms, and sepal width, that is, the average width of the sepals. If we take a look at a tabular representation of these data, we can see that we have five columns. Species, containing the names of the species of Iris flower, petal length in centimeters, petal width, sepal length, and sepal width. This data set is very popular in both statistics and machine learning. It's simple to understand and useful for a wide variety of statistical modeling and machine-learning applications. So we're going to be using this data set for the remainder of our demos in this course. For our statistical modeling demo, we're going to be creating a simple linear regression model, based on the petal lengths and widths contained in the ris data set. We'll attempt to answer two questions using these data. First, how are petal length and petal width related to one another? We'll create a linear regression model of this relationship to attempt to answer this question. Second, can we use a linear regression model to predict new, unknown petal widths given known petal lengths? That is to say, given the iris data set in our simple linear regression model, can we make reliable predictions about the petal width of a flower that we haven't seen yet, given the flower's petal length? We'll begin by loading the Iris data set. The Iris data set, like the Motor Trends cars data set, comes preinstalled with R. We can load any of these sample data sets simply by using the data function and passing in the name of the sample data set as an argument. When we execute this function, the iris data set will be loaded into memory and assigned to a data frame called iris. Let's take a quick peek at the first few rows of data using the head function. As we can see, we have five columns, Sepal. Length, Sepal. Width, Petal. Length, Petal. Width, and Species. Next we'll create a scatterplot of our two variables of interest, that is, Petal. Length and Petal. Width. We'll do this so that we can provide a visual explanation for our linear regression model as we construct it. We'll create this plot using the base plotting system, which, as you remember, is one of three plotting systems in R. We'll use the plot function, set our x parameter to Petal. Length, set our y parameter to Petal. Width, set a main chart title, set an X axis label, and set a Y axis label. As we can see, this produces a scatterplot with Petal Length on the X axis, Petal Width on the Y axis, and a plot symbol, which in this case is a circle for each observation of an Iris flower whose location corresponds to its Petal Length and Petal Width. Our goal is to create a simple linear regression model that fits these data points. Visually, that means that we want to draw a line of best fit through the center of this set of points. Next, we'll create our simple linear regression model. We'll do this using the linear model, or lm function, set the formula to Petal. Width as a function of Petal. Length. Formulas in R are defined as our outcome variable as a function of our explanatory variable. In other words, y as a function of x. We use the tilde character to represent as a function of in our formula notation. In addition, we can also

create much more complex formulas with multiple explanatory variables and conditioning variables as well. However, we'll defer an explanation of these more complex formulas to a more in-depth course on statistical modeling. Then, we'll set our data parameter to our iris data frame and we'll assign this linear regression model to a variable called model. Now let's summarize this model to see what we've created. We'll use the summary function and pass in our model as an argument. There's a lot of information available in the summary of our linear model. All of this information would be useful if we were trying to determine how reliable our linear model is. However, to keep things moving along, we're just going to focus on the two most important pieces of information in this summary, that is, the y intercept and the slope of our linear function. In the coefficient section of our summary, we have two parameters. First, we have the estimate of the y intercept of this linear function. It's estimated at -0. 36, which means that our regression line will pass through the Y axis origin where x is equal to 0 at -0. 36 on the Y axis line. Second, we have the slope for this linear function, which is labeled as Petal. Length. It's estimated at 0. 41, which means that for every 1-cm increase in Petal. Length, we should expect to see a 0. 41-cm increase in Petal. Width. These two parameters alone describe our linear function, that is, the equation for a line that best fits these data points. To understand our linear regression model better, let's visualize this linear regression line by drawing it on the surface of our scatterplot. We'll do this using the lines function, set our x-parameter to Petal. Length; set our y-parameter to the fitted value of our linear regression model, that is the predicted Petal. Width given each Petal. Length, set the line color to red, and increase the width of the line to 3 pixels to make it easier to see. As we can see, this draws our linear regression line on top of our scatterplot. In addition, this linear regression line appears to fit these data very well. Before we attempt to predict new values with our linear regression model, let's take a look at the correlation coefficient of these two variables. We'll do this using the correlation function, set x to our Petal. Length variable and set y to our Petal. Width variable. As we can see, the correlation coefficient is 0. 96. This means that there's a relatively high correlation between Petal Length and Petal Width. This high correlation, combined with the goodness of fit of our linear regression model, tells us that Petal. Length will likely be a good predictor variable for Petal. Width. So now let's try and use this linear regression model to predict some new, unknown values. To predict new values with our linear regression model, we use the predict function, set our prediction object to our linear regression model, then, set the new data parameter to a data frame containing three new Petal. Lengths that we haven't seen before. Our goal is to try to attempt to predict what the corresponding Petal. Width will be for each new Petal. Length. We'll arbitrarily choose 2 cm, 5 cm, and 7 cm for our new Petal. Lengths. As we can see, the predict function returns a vector of three results, that is, one value for each prediction. For a Petal Length of 2 cm, our linear model predicted a Petal Width of 0. 46 cm.

For a Petal Length of 5 cm, it predicted a Petal Width of 1. 71 cm. And for a Petal Length of 7 cm, it predicted a Petal Width of 2. 54 cm. As we can see, our simple linear regression model does reasonably well at predicting unknown Petal Widths given new Petal Lengths. There's quite a bit more to learn about simple linear regression, like analyzing the goodness of fit of our regression model and plotting its residuals. In addition, there are significantly more powerful statistical models that we can create using R. However, we want to keep things moving along with our topics so that we can cover several of the most important features of using R for data science. If you're interested in learning more about statistical modeling using R, though, I'll point you in the right direction at the end of the last module of this course.

Summary

In this module, first, we learned about statistical models and how they can be used for inference and prediction. In addition, we learned about simple linear regression models. Then, we saw a demo where we learned how to create a simple linear regression model using R. In the next module, we'll learn how to handle big data scenarios using R.

Handling Big Data

Introduction

Hi, I'm Matthew Renze with Pluralsight, and welcome back to Data Science with R. In this module, we'll learn how to handle big data using R. As an overview of this module, first, we'll learn about big data, what it is, and why it's important that we learn how to handle it in R. Then, we'll see a demo where we'll learn how to use a big data extension package called FF, which is used to handle data sets that are too large to fit into memory. We'll also use another big data extension package called Big LM, which is used to create linear regression models for large data sets. This will prepare us to use R to train machine-learning models using very large data sets. So let's get started.

Handling Big Data

First, we need to define what the term big data means. Big data are data sets that are of a volume, velocity, or variety that is beyond the limitations of conventional data processing tools.

By volume, we mean that the size of the data set being analyzed is so big that it can't fit into memory for analysis. Or, the entire data set is so large that it can't fit on a conventional hard drive. By velocity, we mean that the data cannot be processed in a reasonable amount of time, or needs to be processed in near-real time rather than as traditional, daily batch operations. By variety, we mean that the data are no longer just structured as tabular data. We also have unstructured data, for example, text images and audio, and semi-structured data, for example XML and JSON. To work with these data for analysis and prediction, we need to extract relevant features from these unstructured and semi-structured data types. While all three aspects of big data are important, we're going to be focusing mostly on the volume and velocity aspects of big data, as these are the situations you'll most likely encounter while working with your data using R. Big data however, is a moving target, since desktop computers and servers are getting faster, more memory, and larger disks, what classifies as conventional computing is always changing. So what is big data today will not be big data in the future, so it's important to recognize whether our data are growing slower than the hardware's improving, which is a temporary big data problem, or if our data are growing faster than technology's improving, which is a permanent big data problem. We need to keep this in mind when deciding how to handle Big Data scenarios in R. Sometimes we can just purchase or temporarily provision more powerful hardware to solve a temporary big data problem; other times, we have permanent big data problem and need to invest in a much more complex and costly distributed computing architecture, like Hadoop. So how do we know if we have a big data problem or not? I'm going to over-simplify this quite a bit, but essentially, you need to ask yourself a few simple questions. First, can all of the data you need to analyze at once fit into the memory of a single, modern desktop computer? This includes both fitting the data into memory and having sufficient working space for memory for operations being performed. Second, can the entire data set fit on a single, modern hard drive? Are you able to store the whole data set on a single hard drive, or would it span multiple conventional hard drives? Third, can you process all of your data in a few hours, or would it take multiple days to complete? Extremely large data sets can sometimes take days to months to completely process on a single, modern desktop computer. Finally, does all the data fit into tables, or does it involve unstructured data, like images, photos, and audio that must be analyzed as well. If you answer yes to all of these questions, then you most likely don't have a big data problem. However, if the answer to any or all of these questions is no, then you may have a big data problem. To attempt to put things into more concrete terms, let's take a look at this big data decision table. Since big data is a moving target and because technology is changing on a regular basis, this table is just a rough approximation. I've tried to make it as future-proof as possible, however, it will inevitably become out of date after a few years. When deciding if we have a big data problem and how to

handle it, we have three classes of data, small, medium, and big. Small data typically consists of less than a few gigabytes of data. This typically equates to less than a few million rows or so of data. However, the size of a row of data changes from one data set to the next, so this is just a very rough approximation. This would include employee data, sales transactions, support tickets, and other small, tabular data sets. Most desktop computers in 2016, that is the year this course was created, contain a few gigabytes of working memory. So, we can store the entire data set in memory and process these small data sets with R using standard tools on a standard desktop computer. Next, we have medium-size data. Medium-size data typically contain less than a few terabytes of disk space, typically less than a few billion rows. Examples of medium-size data would include web clickstream data, image data sets, and large transactional databases. Most desktop computers in 2016 possess hard drives with a few terabytes of capacity. So, we can handle medium-size data problems using R on a standard desktop with the aid of medium-size data extensions, like FF and Big Memory, which we'll discuss next. Finally, we have big data, which typically involves several terabytes, or even petabytes worth of data. This equates to several billion to trillions of rows of data. Once we have so much data that we can no longer fit it on a single conventional hard drive, we must distribute our data across multiple hard drives using data storage virtualization technologies, like RAID, that is, a redundant array of inexpensive disks, or multiple computers using distributed computing technologies, like Hadoop, where each computer holds a portion of the data and we query the data spread out across the computers as if it were a single computer. To work with big data on distributed systems, we need to use R with big data extensions, which allows us to execute R code across clusters of computers using technologies like Hadoop. Depending upon the class of data that we have, that is, small, medium, and big, and the specific problem we're trying to solve, for example, storage versus performance issues, we have a variety of ways to solve these issues in R. First, the simple solution to handling big data is to throw more hardware at the problem. Oftentimes it will cost less to purchase more powerful hardware than to invest the time, energy, and resources to build and maintain a big data solution. In addition, it's now super easy and relatively inexpensive to provision an extremely powerful virtual machine in the cloud just to process your data, and then de-provision it once the analysis is complete. In fact, I use this technique on a semi-regular basis to handle processing big data jobs in the cloud that are beyond the capabilities of my desktop computer. Second, if more powerful hardware isn't an option, we could also work with just a subset of the data. If we're connecting to a database with R, we can use a SQL query that will select just the columns and rows that we need for our analysis. Then, we only load the necessary columns and rows into memory on our desktop machine. Oftentimes, this will reduce our big data problem down to a more manageable size. If subsetting our data isn't sufficient to make it manageable, we may need to sample the

data instead. Sampling is a technique where we randomly choose a smaller subset of rows from our larger data set. When we sample our big data though, we're kind of defeating the purpose of collecting and maintaining such a large set of data. However, this is often a very low-cost solution to the problem if you're in a bind and you can't afford a better solution. Fourth, if your big data problem is limited by performance rather than storage, Microsoft has a free, enhanced, 64-bit distribution of R called Microsoft R Open. It allows you to use multiple threads, rather than a single thread, like the native R distribution we're using in this course. Depending upon the number of cores in your CPU, it can significantly improve the performance of many operations involved in statistical analysis and machine learning. Fifth, if your data set is too big to fit into memory, but still small enough to fit on your hard drive, you can use one of a few medium-sized data packages, like FF or Big Memory. Both FF and Big Memory take a large data file on disk and process it piece by piece in smaller chunks that are small enough to fit into memory. In addition, there are numerous extension packages that provide memory- optimized and performance-optimized versions of the standard statistical operations and machine-learning algorithms. For example, if we need to create a linear regression model on a large data set, there's a package called Big LM that provides a bounded-memory version of the linear regression algorithm to prevent the algorithm from running out of memory on large data sets. We'll be looking at both FF, which stands for flat file, and Big LM, which stands for big linear model, during our upcoming demo. Sixth, if we have big data, that is, a data set that's too large for both our memory and hard drive, or cannot be processed by a single desktop computer in a reasonable amount of time, we have third-party, big data extension packages to solve this problem. For example, we have PDBR, which stands for programming with big data in R, which is a set of open source extension packages for using R with large scale, high performance distributed computing environments. It allows us to split up our big data set across several computers, which each store a small piece of the data set and then work together to perform R operations on their respective pieces of the data set as if they are one single unified data set. In addition, there are other open source R extension packages for working with other open source distributed computing technologies, like Hadoop, HDFS, H-base, Map Reduce, Hive, and Avro. Finally, we have Microsoft R Server, formerly known as Revolution R Enterprise. Microsoft R Server overcomes the processing, memory, disk, and network constraints of handling big data with R, using multithreading, distributed computing, and data source connectors. It currently supports Linux, Terandata, Hadoop, and soon HDInsight. In addition, there's a version of Microsoft R Server built inside of SQL Server 2016, as well. However, these are proprietary technologies and are not free, open source software. Now let's see a demo where we'll learn how to use FF and Big LM to handle a big data scenario in R.

Demo

For a big data demo, we're going to use the Iris data set again. While it would be nice to set up a Hadoop cluster and process a few terabytes of data for our demo, unfortunately the amount of time to set up and walk through a demo of this complexity would be too much for a beginner course. In addition, most viewers of this course will probably not be connecting R to a distributed computing environment. However, they will likely run into data sets that are larger than the memory of their desktop computers. So instead, we'll keep things simple, yet interesting, and still widely applicable for this demo. We'll do this by demonstrating how to use the big data tools FF and Big LM, using a smaller, and much easier to download data set like the Iris data set. This means that we'll have to imagine that we're working with a much larger data set than the actual data set that we'll be using in the demo. However, the results will essentially be the same as if we were actually using a very large data set. In this demo, we'll attempt to answer the same questions we answered during our previous demo. That is, how are petal length and petal width related, and can we predict a new, unknown petal width given a known petal length. However, keep in mind that we're going to pretend that our Iris data set is much larger than could fit into the memory of our computer, and use the big data extension packages FF and Big LM to perform the same tasks as our previous demo, in a way that would allow us to work with a much larger data set. First, let's set the working directory to a folder containing our Iris data set. Next, we'll download, install, and load the FF package. Then, we'll create an FF data frame that we'll call irisff using the read. table. ffdf function. Note that the df in ffdf stands for data frame. We'll set our file to a file called Iris. csv that I created from exporting the Iris data set to a CSV file. And we'll set our Read function parameter to read. csv since we'll be reading from a CSV file. With FF, our data must be stored on disk in a flat-file format, for example, a CSV file format. FF can then read data from this flat file into memory, piece by piece, rather than having to load it all into memory at once. Please note that the read. table. ff function is not reading the data from the Iris. csv file into memory. Rather, we're creating a pointer to the data in the file so that we can access it piece by piece as if it were a data frame stored in memory. To verify this, let's use the class function on the irisff variable we created. As we can see, it returns an object of type ffdf, rather than a data frame like we've seen in the past. However, we can still use many of the same functions that we use on a data frame using an ffdf object. For example, we can get the names of our variable using the names function, which as we can see returns the five column names in our iris data set. In addition, we can inspect the first few rows of data from our data frame as well. However, we can't use the head function to do this since we're working with an ffdf object now, rather than a data frame. Instead, we need to use the indexing operators like we did in module 1. As we can see, this

returns the first few rows of data. So, there are some functions that will work the same with both a data frame and an ffdf, and others that do not behave the same. Getting to know which functions behave differently and potential workarounds is an important step to learning how to use FF. Now let's create a linear model for our hypothetical big data. First, we need to download, install, and load the Big LM package. Big LM is a bounded memory version of the standard linear regression model. This means that we can create a linear regression model for large data sets using significantly less memory than the traditional linear regression algorithm. We create our model using the biglm function, set our formula to Petal. Width as a function of Petal. Length, and set our data to our irisffdf object. The ffdf object will feed data into memory for the biglm algorithm, piece by piece, to prevent R from running out of memory if our source data were actually larger than our computer's available memory. And the biglm function is going to use significantly less memory to build our linear model, which also helps us work within our memory constraints. Now let's inspect the model using the summary function. As we can see, we have the same values for our y intercept and our slope parameters as we did during our last demo. Essentially, we can produce the same linear model with a much larger data set. In addition, just to see that we've, in fact, created the same linear regression model, let's visualize our model again. First, we'll create a scatterplot of our Iris data set using the same function and parameters we used during the last demo. However, to access the data contained in a column inside the ffdf object, we need to place square brackets at the end of our reference to the Petal. Length and Petal. Width variables. This is because those column names are just a reference to the columns of data in the CSV file and disk, not the actual vectors of data themselves. It's yet another difference that's important to learn when using and ffdf, instead of an in-memory data frame. Next, let's draw a linear regression line on the plot. Since the object that's returned from the biglm function doesn't contain a property for the fitted values of the model, like our regular linear regression model did, we'll have to construct the line ourselves using the y-intercept, the slope, and the equation for our line. First, let's get the y-intercept from our linear model. We'll do this by getting a summary of the model, accessing the matrix from the model's estimated parameters, and getting the value contained in the first row and first column, which is the estimate for the y-intercept. We'll assign this value to a variable called b, which is standard notation for the y-intercept in the equation for our line. Next, we'll get the slope from the second row and first column of the matrix and assign it to a variable called m, which is standard notation for the slope in an equation for our line. Now, we'll draw our linear regression line on the scatterplot using the lines function, set our x-parameter to Petal. Length values and our y-parameter to the equation for our linear regression model, that is, y = m * x + b, or in simpler terms, the Petal. Width is equal to the estimated slope, multiplied by the Petal. Length, plus the estimated y offset, set our line

color to red, and set our line width to 3. As we can see, we've drawn our linear regression line on the scatterplot of the iris data set. Finally, let's predict a few unknown petal widths given known petal lengths. We'll do this using the same function and parameters as we did in our previous demo. However, we need to make one slight modification. We need to add a Petal. Width column to the data frame we assigned to our new data parameter, and set the value to 0 for each row in the data frame. This is because Big LM requires a column for our outcome variable in order to make its predictions. It's unfortunate that this discrepancy exists between the two linear model prediction methods, however, it's a simple enough fix. As we can see, we predicted the same petal width values given 2-cm, 5-cm, and 7-cm petal lengths. There are many more tools and extension packages available for handling all sorts or big data scenarios in R. Covering them all in a single module would not be possible. However, I'll point you in the direction of resources to learn more about handling big data in R at the end of the last module of this course.

Summary

In this module, first, we learned about big data and why it's important to learn how to handle big data in R. In addition, we learned about several tools and extension packages that allow us to handle big data scenarios. Then, we saw a demo where we learned how to use FF and Big LM to create a linear regression model with big data. In the next module, we'll combine our knowledge of statistical modeling in big data to learn how to make predictions using machine learning with R.

Predicting with Machine Learning

Introduction

Hello again, and welcome to our next module on Data Science with R. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to make predictions using machine learning with R. As an overview of this module, first, we'll learn about machine learning and how it can be used to make predictions about our data. We'll cover the various types of machine-learning algorithms and how to train and evaluate our machine-learning models. Then, we'll see a demo where we'll

apply what we've learned by creating a decision-tree classifier and make predictions using this classifier. So let's get started.

Predicting with Machine Learning

First, what is machine learning? Machine learning is a subfield of artificial intelligence. The goal of artificial intelligence is to create machines that act rationally in response to input in their environment and make decisions to achieve a goal of some kind. Machine learning is artificial intelligence where the machine learns to complete tasks without explicitly being programmed step by step to complete those tasks. Instead, we use data to train the machine-learning model how to complete the task, instead of having a human write code to complete the task instead. These tasks can include detecting a person's face in a photograph, predicting tomorrow's temperature, driving a car, along with many other tasks. Essentially, with machine learning, we're trying to use existing data to learn a function that maps new data to a prediction. For example, imagine that we want to create a function to determine whether a photo contains an image of a cat or not. First, we would need to create a data set that contains images with cats and images without cats. We would have a human being label each photo indicating whether it contained a cat, or not. Next, we would apply a machine-learning algorithm to the data set of images with and without cats to have it learn a function that predicts whether an image contains a cat or not. Finally, if we've done everything correctly, we should be able to provide the function with a new image and it will tell us whether it contains a cat or not. While this is a vastly over-simplified explanation of machine learning, it essentially captures the essence of what we're attempting to accomplish. Machine-learning algorithms prefer a very large amount of data in order to train their models to make accurate predictions. So we typically need lots of high-quality data to build accurate machine-learning models. All modern machine learning is based on statistics, so we make heavy use of statistical methods and theory. In many ways, statistical modeling and machine learning are very similar to one another. In addition, machine learning is also very similar to data mining, as well. In fact, depending on who you ask, some people will say that machine learning, statistical modeling, and data mining are essentially the same thing. However, others will point out differences in terminology, the amount of data involved, the abstractions involved, the complexity of the models, or differences in the primary goal or outcome of each of these three similar fields. Because machine learning is built on statistics, we use statistical models in most of our machine-learning algorithms. Like statistical modeling, with machine learning, our models are typically composed of a set of parameters. In machine learning, we use a training algorithm to estimate, that is, to set these parameters. This is a process referred to as training the model. The

algorithm uses existing data called training data to set the parameters for the machine-learning model. Then, we use the model to make predictions given entirely new data that the model has never seen before based on the probabilities contained in the machine-learning model. So what can machine learning actually do? There are essentially four main types of predictions that machine-learning algorithms can attempt to make. First, we have classification, where we attempt to predict categorical values, that is, qualitative values. For example, trying to predict whether a photo contains an image of a cat or not, or trying to predict the species of flower based on its size, shape, and color. Next we have regression, where we predict numeric values, that is, quantitative values. We performed a very basic type of regression, that is, simple linear regression, in the demos in the last two modules of this course. Other examples of regression include predicting what the average temperature will be tomorrow, predicting the average fuel economy of a new car, or predicting the price that a house will sell for. Then, we have clustering where we find groups of similar observations in our data. Examples include trying to group flowers by similar sizes, shapes, and colors; recommending which movies we might like, based on others who have viewed similar movies to us; and segmenting our customers into various groups, based on demographics. Finally, we have anomaly detection where we find data that are abnormal, called outliers in our data set. Examples include detecting credit card fraud, intrusion detection in computer networks, and predicting possible equipment failure. Most of the complex and interesting things that machine-learning algorithms can do are essentially built on these four types of predictions. There are essentially three main types of machine learning. First, we have supervised learning, where humans provide output labels, that is, the correct answer for each row of data in our training set. Then, the machine uses these labeled data to learn how to make predictions given new data without the labels, for example, classification, and regression. Next, we have unsupervised learning where humans do not provide any labels and the machine learns the structure of the data or finds patterns in the data, for example, clustering. Finally, we have reinforcement learning, where the machine attempts to achieve a goal in a dynamic environment and receives reinforcement, depending upon whether it's getting closer to achieving its goal or not. We already saw supervised learning, that is, predicting a numeric value when we built our linear regression model in our last two demos. We provided the linear regression algorithm, a correct answer, that is, the petal width for each row of input, that is, the petal length. Then we used our trained linear regression model to predict unknown petal widths, given new petal lengths. During our machine-learning demo in this module, we'll perform supervised learning again, however this time we'll perform classification, that is, predicting a categorical value using a decision-tree classifier. Due to limited time, however, we will not cover unsupervised learning or reinforcement learning in this course. There are so many types of machine-learning algorithms in

existence, that we couldn't possibly cover them all in a single course. In addition, there are new algorithms being developed every day. However, just to give you a bit of exposure so that you might recognize the names of some of these machine-learning algorithms if you hear them mentioned, we'll run through just a few of the most popular types of machine-learning algorithms in use today. Please keep in mind that for each of these general algorithm types, there might be dozens of different, specific implementations. In addition, all of these types of algorithms are available in one form or another in R. We have decision-tree classifiers, Naive Bayes classifiers, linear regression, support vector machines, neural networks, k-means clustering, and ensemble learning, where we use a collection of simple machine-learning algorithms together. We can mentally organize all of these algorithms into either supervised or unsupervised learning, and by which of the four types of predictions they are making, that is, classification, regression, clustering, or anomaly detection. The machine-learning process is a bit complex, but we're going to simplify it and walk through the process step by step. First, we start with a set of data that we're going to use to create our machine-learning model. We'll assume that these data have already been cleaned and transformed into a format necessary for machine learning. If this is supervised machine learning, which is a type of machine learning we're interested in for our demo, our data set will have all the explanatory variables in the outcome variable we are trying to predict, prepopulated. Next, we split our data into two sets, that is, a training set and a test set. We'll use the training set to train our machine-learning model and we'll use the test set to verify how accurate the predictions of our model are. There are numerous methods we can use to split our training and test data, including methods like cross-validation. In addition, it's often advisable to split our original data set into three data sets, that is, a training set, a validation set, and a final test data set. However, we'll just keep things simple during our upcoming demo and we'll use a random sample to split our original data set into a training data set and a test data set. Next, we feed our training data set into our machine-learning algorithm. The algorithm will use these data to create a machine-learning model. Once we have a model with our parameters set, we feed the test data into the machine-learning model to see if its predictions match the answers in the test data set. We refer to this part of the process as validation. This allows us to verify the model's prediction accuracy and several other measures of the model's predictive capabilities. In the real world, we often go through numerous iterations of training and testing various machine-learning models and fine-tune the model to produce a more accurate and robust model. Once we have a machine-learning model with a predictive power suitable for our needs, we deploy it to production and begin feeding it production data. These data will contain all the explanatory variables, but will not contain the outcome variable, that is, the variable we're attempting to predict. The machine-learning model then uses the explanatory values contained in the

production data and makes a prediction and returns the results of this prediction. I've taken the liberty of simplifying this process quite a bit to make it easy to explain in a short period of time. There's a lot more work involved in model selection, model tuning, avoiding overfitting, and creating robust models that we've excluded for the sake of brevity. However, this should give you a basic idea of how the machine-learning process works. We only have time to learn about one machine-learning algorithm in this module. In addition, we already learned about regression, that is, predicting a numeric variable during our linear regression demo. So now, we're going to learn about classification, that is, predicting a categorical value. To keep things simple, yet interesting, we'll learn about a type of classifier called a decision-tree classifier. A decision-tree classifier is a supervised-learning algorithm, that is, a human assigns the correct answers to the training data set during the training phase of the algorithm. A decision tree models a decision making process as a set of decisions, that is, we have a series of nodes, one for each question, and branches, one for each possible answer. However a machine, rather than a human, creates this decision tree based on probabilities in a concept from information theory called information gain. Decision trees are probably the easiest machine-learning algorithm to understand. In addition, they're also very transparent, that is, there's no mystery in determining how they made a specific prediction. So they're very popular in cases where simplicity and transparency is necessary. The decision tree on the screen is based on data contained in the passenger manifest of the R. M. S. Titanic, the British passenger ship that sank in the north Atlantic on the morning of April 15, 1912, after colliding with an iceberg. This decision tree predicts whether a person would be more likely to survive the sinking of the Titanic, or die during this tragedy, based upon three predictive variables contained in the data set, that is, gender, age, and the number of family members on board. And one outcome variable, that is, whether they survived or whether they died. This decision tree models the decision-making process leading to the prediction as a tree of three questions. First, the decision tree asks if the person was male. If the answer is no, that is, they are female, it predicts that they will survive based on the statistical likelihood that females survived the sinking of the Titanic. If the answer is yes, that is, they are male, it asks a second question. Is the male's age greater than 9. 5 years old? If the answer is yes, it predicts that they will die based on the high likelihood that men over the age of 9. 5 died on the Titanic. If the answer is no, that is, they were younger than 9. 5, then we ask a third question. Does this male have more than 2. 5 family members on board? Since we can't have half-siblings, half-spouses, or half-children, we'll just round this number up to three. If the answer is yes, that is, if the male child has three or more family members on board, we predict that they will die. However, if the answer is no, that is, if they do not have three or more family members on board, we predict that they will survive. This decision tree was constructed entirely by a machine, solely from the data contained in the

Titanic's passenger manifest, and whether the person survived or died during the sinking of the Titanic. It does a very good job predicting statistically who would have lived or died, based upon these data. Decision trees are a very simple, yet powerful form of machine learning. So now, let's learn how to create a decision-tree classifier and use it to make predictions using R.

Demo

For our machine-learning demo, we're going to use the Iris data set again. We're going to attempt predict the species of Iris flower given the petal length, petal width, sepal length, and sepal width. In addition, we also want to know how accurate is our model's ability to predict the correct species. We'll attempt to make these predictions using a decision-tree classifier. First, let's load the Iris data set into memory using the data function. Next, since machine learning involves using randomly generated numbers during the process of building models, we need to make a sequence of randomly generated values, reproducible by setting a random number seed. By setting random number seed to an arbitrary number, R will produce the same sequence of random numbers every time this script is run on any computer. This means that both you and I will create the exact same machine-learning model if we run this script on our respective computers. I've semi-arbitrarily chosen the number 42 as a random number seed as an homage to Douglas Adams, the Hitchhiker's Guide to the Galaxy. However, the number that we choose is irrelevant as long as we both use the same number for our random seed. Next, we'll split our iris data set into both a training data set and a test data set. We have 150 rows in our iris data set and we want to use about two-thirds of the rows to train our model, and the remaining one-third to test our model's prediction accuracy. So, we'll create a training data set containing 100 randomly sampled rows from our iris data set, and a test data set containing the remaining 50 rows. We'll do this by creating a vector of 100 randomly generated indexes for our 150 rows in the iris data set. We'll use the sample function and set our x parameter to a sequence of numbers ranging from 1 to 150. These numbers will represent the index of each of our 150 rows of data in the iris data set. Then, we'll set the size of our sample to 100 indexes, since we want to create a training data set containing 100 rows, and we'll leave 50 rows behind for our test data set. Next, we'll assign this random sample of 100 indexes to a variable called indexes. Now, let's take a quick look at the values contained in our indexes variable to see what we've created. As we can see, we have 100 randomly selected row indexes, which will correspond to the index of each row we want to use in our training data set. Next, we'll use our 100 randomly sampled indexes to create our training data set. We'll create a subset of our iris data frame by passing in the indexes of the rows we want to sample into the row index argument, and leave the column index argument blank

since we want all columns in the data frame. We'll assign the subset of the iris data frame to a variable called train. We'll do the same for our test data set; however, we'll set the row index argument to the row indexes variable, but we'll prefix it with a minus sign. This tells the row index argument to include all rows, except for the ones listed in the row indexes variable. Essentially, this returns the other 50 rows in the iris data set. We'll assign these rows to a variable called test. Next, we'll create our decision-tree classifier. First, we need to install, download, and load the decision tree package using the library function. Next, we create a decision-tree model using the tree function, set our formula to Species as a function of all other variables in the data set. We do this using the notation Species tilde dot. The dot tells R to include all other variables in the data frame as our explanatory variable. We could have gotten the same result by typing species, tilde, Petal. Length + Petal. Width + Sepal. Length + Sepal. Width. However, using the dot notation is simpler in this case. We'll set our data parameter to our training data set and we'll assign our model to a variable called model. Next we'll inspect the model using the summary function. As we can see, our decision tree only used two of the four explanatory variables in the decision tree. Since these two variables alone were sufficient to create the best decision tree given the data in the algorithm. In addition, we can see that the decision tree contains 4 terminal nodes, that is, there are 4 leaves in the tree. For now, we won't worry about the other two values that are model reports. Next, let's visualize our decision-tree model. We'll do this using the plot function and pass in our model as an argument. In addition, we'll add text labels to our tree as well. As we can see, this creates a tree diagram representing the decision-making process that our decision-tree model will use to predict new values. First, if the petal length is less than 2. 6 cm, we'll just assume it's a setosa and move on. However, if the petal length is greater than or equal to 2. 6 cm, then we'll move on to the next branch in our decision tree. Next, if the petal length is greater than or equal to 4. 95 cm, we'll predict that it's a Virginica; otherwise, we'll go to the next branch in our decision tree. Finally, if the petal length is less than 1. 65, we'll predict that it's a versicolor; otherwise we'll predict that it's a Virginica. In addition to visualizing our decision-tree model as a tree diagram, we can also visualize the decision boundaries of our decision tree on a two-dimensional scatterplot of Petal Length and Petal Width. To do so, we're first going to load ColorBrewer to create a color palette that's accessible to individuals with certain types of colorblindness. Next, we'll use ColorBrewer to create a palette of 3 colors from the Set2 color palette. Now, we'll create a scatterplot of Petal. Length on the X axis, Petal. Width on the Y axis, and we'll color each plot symbol by species. Finally, we'll plot the decision-tree boundaries on top of the scatterplot using the partition. tree function, set the tree parameter to our decision-tree model, set the label parameter to our Species variable, and set the add property to TRUE, indicating that we want to add these decision boundaries to our previous scatterplot. As we can

see, this allows us to visualize our decision tree by showing us the boundaries of each decision. This provides the same information as our decision-tree diagram, but in a way that might be easier for some people to comprehend. Next, let's verify the prediction accuracy of our machine-learning model by comparing its predictions to the correct answers in our test data set. We'll do this by using the predict function, set the prediction object to our tree model, set the newdata parameter to our test data set, and set the prediction type to class for classification, then we'll assign these predictions to a variable called predictions. Next, we'll use our predictions and the known correct answers contained in our test data set to create what's referred to as a confusion matrix. We'll create it using the table function, set x to our predicted values, and set y to our correct answers contained in our test data set. This creates a two-dimensional matrix showing us how many predictions were correctly classified versus how many were incorrectly classified. On the rows, we have our predicted species; on the columns we have our correct species. As we can see, we correctly classified all 17 setosa flowers in our test data set. In addition, we correctly classified all 15 virginica flowers in our test data set. We correctly classified 16 out of the 18 versacolor flowers; however, we misclassified 2 versicolor flowers as virginica flowers. All in all, this decision-tree classifier appears to be pretty accurate given these results. While a confusion matrix is useful to see how the machine-learning model either correctly or incorrectly classifies your data, we typically need much more information to determine how accurate or reliable our machine-learning model is. To get this information, we'll download, install, and load the caret package, which stands for classification and regression training. Then, we'll use the confusionMatrix function in the caret package, set the data parameter to our predictions, and set the reference parameter to the correct answers in our test data set. As we can see, this creates the confusion matrix we just saw, in addition to a significant amount of information about our machine-learning model's predictive capabilities. When creating machine-learning algorithms in the real world, all this information is useful for evaluating your model's predictive capabilities in various ways to optimize it for various objectives. However, to keep things simple for this course, we're just going to look at the prediction accuracy of our decision-tree model. As we can see, our decision tree predicted the correct species of flower for 96% of the flowers in our test data set. This is a relatively high prediction accuracy with such a simple machine-learning algorithm. Given more training data, more powerful machine-learning algorithms, and better techniques for fine-tuning our machine-learning model, we could get this prediction accuracy even higher. However, we'll have to defer this to a more in-depth course on machine learning. At the end of the last module of this course, I'll point you in the direction of additional sources of information to learn more. Finally, we're going to save this decision-tree classifier model so that we can use it to make predictions with new data in our final demo in the next, and final, module. First, let's reset the

working directory just in case it hasn't been set yet. To save an R object to the file system so that we can use it later, we use the save function and pass in the object we want saved as an argument. The R object will be saved as a file to the working directory we specified using the setwd function. Machine learning is a very powerful aspect of data science. In fact, it's probably the number one reason why data science has become so popular in recent years. There's so much more that I'd like to teach you about machine learning, including how to select appropriate algorithms, how to fine-tune our models, how to validate the accuracy and reliability of our models, and the latest breakthroughs in machine learning, like deep-neural networks. However, we'll have to defer all of this information to more in-depth courses on machine learning with R. I'll point you in the direction of these more advanced courses at the end of the last module of this course.

Summary

In this module, first, we learned about machine learning and how we can use it to make predictions. We covered various machine-learning algorithms and how to train and evaluate our model. Then, we saw a demo where we learned how to create a decision-tree classifier to predict the species of flower given its dimensions. In the next module, we'll learn how to deploy our R applications into production.

Deploying to Production

Introduction

Hello again, and welcome to the final module in our introduction to Data Science with R. I'm Matthew Renze with Pluralsight, and in this module we'll learn how to deploy R into production. As an overview of this module, first, we'll learn how to deploy R into production. We'll learn about the various ways we can share data, code, and analyses with others. In addition, we'll learn about a web-based, interactive application framework for R called Shiny. Then, we'll see a demo where we'll learn how to convert our machine-learning model from our last demo into a web-based application in R using Shiny. Finally, we'll wrap things up for the course as a whole. In addition, I'll show you where to go to learn more about all the topics we discussed in this course.

Deploying R into Production

So far, we've learned how to do some pretty impressive things with R. We've created scripts to transform and clean data, we've analyzed data using descriptive statistics, data visualizations, and statistical models, and, we've made predictions with data using big data and machine learning. However, oftentimes we need to share our work with others. In general, we want to put our R code into production. There are several ways we can do this in R. First, we can create documents using R Markdown with tools like Sweave and knitr. R Markdown is a mini-language like HTML, but much simpler, that we can use to create documents, for example, web pages, Microsoft Word documents, and PDFs, that contain both our R code and the results of our analysis. It seamlessly blends our code, text output, data visualizations, and documentation into a single document. Second, we can create interactive data applications using frameworks like Shiny. Frameworks like Shiny allow us to put our R code on the web so that users can interact with the data and see the results. We'll be learning more about Shiny next and using it during our upcoming demo. Third, we can deploy our R code to on-premise servers using technologies like Microsoft R Server and DeployR. This allows us to run our scripts to process data and to produce output like data visualizations that can be embedded into online documents, reports, and applications. Finally, we can deploy R code to the cloud using tools like Microsoft Azure Machine Learning. Despite the fact that Azure Machine Learning is primarily designed for creating machine-learning applications, it's currently one of the easiest ways I'm aware of to take an R script and deploy it to the cloud. It automatically creates a web service for your R script, given the expected inputs and outputs, so that any developer can run your R script, simply by coding to the web service's RESTful API. Expect to see many more ways to deploy R into production in the upcoming years. This is a really active space with a lot of initiatives moving things forward. However, since we don't have time to learn about all these technologies, we're just going to focus specifically on deploying R into production using Shiny. Shiny is an interactive data visualization framework made by the creators of R Studio, that is, the IDE that we've been using throughout this course. Shiny applications are composed of two main components, that is, a user interface and a web server. A user interface is used to present a series of widgets, that is, input and output controls, that take input from and present output to the user. The web server takes input from the user interface and performs work, and then returns the output to the user interface. All of the code to create a Shiny application is written in R. In addition, the Shiny Team has made it very easy to create web-based, interactive data applications using Shiny. For these reasons and more, Shiny has become very popular within the R community in the past few years. We begin creating Shiny applications by creating a layout for the user interface. All layout elements are created using layout functions. There are numerous layout functions to create the various layout elements. However, for our demos in this course, we're just going to focus on the five most important layout elements. For example, the code on

the left would produce a layout that looks like this. Starting from the top of the code, we have a fluid page, which creates a page with a fluid layout rather than a fixed layout. Next, we add a title panel, which renders the title at the top of our page. Then, we add a sidebar layout, which creates a layout for both a sidebar and a main area. Next, we add a sidebar panel, which creates the container for our sidebar content. Finally, we add a main panel, which creates a container for our main content. After we've created a layout, we can add a variety of widgets, that is, input and output controls, to our user interface. Shiny widgets include buttons, checkboxes, sliders, and input boxes for dates, numbers, and text. These are just a few of the widgets that we have available in Shiny. To add widgets to our Shiny layout, we add each widget using a function in R. For example, the code on the left creates the following user interface. First, we add a sliderInput widget, set the inputId, which is the unique identifier for the control to an Id called number, set a label with the character string Choose a Number, which is rendered as a label for the control in the UI, set the minimum for the slider to 0, set the maximum for the slider to 100, set the value to 25, which sets the initial value of the slider. Finally, we add a text output widget and set the output Id to text. The text label on the right of the UI currently shows the text You selected 25, however, in order to set the text of the text output label to the value we see on the screen, we need to create a web server and wire up the UI to the server, which we'll do next. A Shiny web server, like the code on the right, is a function in R that simply takes an input and returns an output. The input comes from the widgets in the UI that we previously created. The output is returned to the widgets in the UI. For example, this server simply takes the value from our input slider, which we gave an Id of number, and concatenates the text You selected with this value using the paste function. Then, it uses the renderText function to render this text and assigns the rendered text to the text output widget we gave an Id of text. Please note that the output variable that is returned from the user interface is actually defined as an input parameter in the signature of the server function. This is referred to as an out parameter in programmer lingo. This is different from most usual functions that return the output of the function as a normal return value. Finally, to run a Shiny application locally on our machine, we need to create a Shiny app. We do this by wiring up the user interface and the web server into a single application using the shinyApp function. The shinyApp function takes as its arguments a shiny user interface and a shiny server. This will launch a new Shiny application in our web browser on our local machine. However, we can easily deploy our Shiny app to our own on-premise web server or to the cloud as well. Now, let's see how to create a Shiny application using R and RStudio.

Demo

For our final demo, we're going to take the machine-learning model that we created in our previous demo and deploy it to production. In addition, we're going to tie together several of the concepts we've learned about throughout this course to build this application. Once again, the application will answer the following questions for our end users. What is the species of Iris flower, based on both the petal length and petal width of a new flower? The user of our application will supply these values through our user interface. In addition, we'll attempt to convey to the user how this prediction was made by creating a data visualization. We'll visualize the user's input, relative to the decision boundaries of our decision-tree classifier used to make the prediction. But first, before we begin creating this application, we need to walk through two simple demo applications using Shiny. We'll create a Hello World app and a basic input/output application so that we can learn the fundamentals of Shiny. To get started learning how to create interactive data applications using Shiny; first, we're going to build a simple Hello World application with Shiny. We'll start by loading the Shiny package into memory. Next, we'll create a very simple UI, composed of a fluid page in the text Hello World. Then, we'll create the simplest server possible, that is, a server that takes input and returns output, but performs no work in the body of the server function. Finally, we'll wire up the UI and the server as a Shiny application on our local machine. As we can see, this produces a Shiny application that displays the words Hello World in our browser. Next, let's create a simple Shiny application that takes a single input and produces a single output. First, we'll create a UI composed of a fluidPage, add a main titlePanel, add a sidebarLayout containing a sidebarPanel, which will contain a sliderInput widget with an inputId of number, a label, and minimum value of 0, a maximum value of 100, and an initial value of 25. We'll also add a mainPanel to our sidebarLayout, containing a single textOutput control with an outputId of text. Next, we'll create a server to process this UI's input and return output to the UI. We'll create a function called server, which takes two arguments, that is, an argument called input and another called output. In the body of this function, we'll set the value of our textOutput widget, whose Id is text, to the results of our render output function. Within this function, we'll concatenate the character string you selected with the numeric value of our slider input, whose Id is number. We've perform this concatenation, that is joining of two or more character strings, using the paste function. This server function is all we need to convert our input to our output on the web server side of the Shiny application. Finally, we need to wire up our Shiny UI and our Shiny server into a Shiny application so that we can run it on our local machine. We do this using the shinyApp function and we pass in a user interface and a server. As we can see, we've just created a simple Shiny application that allows us to change the value of the slider input and see the result of the change in our text output. Now let's take things a step further by creating our interactive machine-learning application in R with Shiny. First, we need to set the

working directory to the directory containing the decision-tree classifier model that we created in our previous demo. Next, we need to load that decision-tree model into memory using the load function and pass in the file name of the file containing our R object as an argument. Our decision-tree classifier model has now been loaded into memory and has been assigned to a variable named model, which was the variable name this object had when we saved it in our previous demo. Then, we'll load the ColorBrewer library so that we can create a user-friendly, and handicap-accessible color palette. Finally, we'll create a color palette so that we can use it later when we render our data visualization. Next, we need to create a user interface. This user interface is going to contain two numeric sliders for our user input, that is, one for petal length and one for petal width. And for output, it will contain a plot of our data visualization and a block of text informing the user of the predicted species. We'll create a fluid page layout with a main title, a sidebarLayout with a sidebarPanel, containing a sliderInput for Petal Length with an Id of petal. length, a label for the slider, a minimum value of 1 cm, a maximum value of 7 cm, and a default value of 4. In addition, we'll create a second numeric slider for Petal Width with an Id of petal. width, a label, a minimum value of 0, a maximum value of 2. 5, a step of 0. 5, which indicates that we can move the slider in half-centimeter increments, and a default value of 1. 5. Within our main panel to the right of our sidebar we'll add a placeholder for a textOutput for our Species prediction results, and a plotOutput for our data visualization of the decision-making process for the prediction. Next, we'll create our Shiny server. This server will take the two slider values, that is, Petal Length and Petal Width as input, and return a prediction result and a data visualization as output. First, we'll set the text output of our results to a renderedText function. In the body of this function, we'll create a data. frame with four columns in one row. Then, we'll assign the Petal Length input that the user provided via the first slider to the Petal. Length value. We'll assign the Petal Width input provided by the user via the second slider to the Petal. Width value. Then, we'll set both the Sepal. Length and Sepal. Width to 0, since the decision tree doesn't need to use them in the actual prediction process. We still have to provide these columns though since the model expects them as input. However, they'll have no effect on the prediction. Next, we'll make our prediction just like we did in our previous demo, using the predict function, setting the prediction object to our decision tree, setting the new data parameter to the single-row data frame we created from the user's input, and setting the type to class since we'll be performing classification. Finally, we'll create a text string that says The predicted species is, concatenated with the predicted species name. For the data visualization portion of this application, we'll start by setting the output$plot equal to the result of the renderPlot function. In the body of this function, we'll create a scatterplot colored by species, just like we did in our previous demo. Then, we'll plot the decision tree boundary, once again, like we did in our previous demo. However, this

time we'll place a red x on the plot indicating the petal length and width that the user selected to show them where their input lies, relative to the decision boundaries. We'll do this using the points function, set the x-parameter to the petal. length that the user selected, set the y-parameter to the petal. width that the user selected, set the color to red, set the plot character to 4, which represents a plot symbol shape like an x, set the plot symbol size to 2 to make it twice as big as the normal plot symbols, and set the line width to 2 to make the line twice as thick. Now, we just need to wire up the user interface and the server into a new Shiny application. As we can see, this creates a web-based, interactive application with two sliders, that is, one for Petal Length and one for Petal Width. In addition, we have a text field for the prediction result and a data visualization showing the user how the prediction was made with the red x at the location corresponding to the user's input. We can drag the first slider to increase Petal Length, which changes our prediction result and the output of our red x indicating the user's input, relative to the prediction boundaries. And, we can drag the second slider to increase Petal Width, which changes both our prediction result and the data visualization as well. And there we have it! We've successfully created an interactive machine-learning application using R and Shiny. While this was a relatively simple application, you can imagine how we could extend what we've already learned to create more complex, interactive, data-driven applications in R.

Where to Go Next

Now that you've had an introduction to Data Science with R, you probably want to learn more so that you can become more proficient with both data science and R. In order to help you with your learning objectives, I've compiled a list of more in-depth resources for each module we covered in this course so that you can choose to learn about whatever topics you found most interesting. First, if you'd like to know more about programming with R, Abhishek Kumar has a seven-hour course on R Programming Fundamentals. It covers variables, operators, data structures, control flow, functions, and more. In addition, Casimir Saternos has a three-hour course on getting started with RStudio. It covers various aspects of both R and RStudio, including the environment, data import, plotting, and key R packages. If you're interested in learning more about working with data in R, I recommend watching my 2. 5 hour course on Exploratory Data Analysis with R. The second module, titled Transforming and Cleaning Data, goes more in depth into the process of importing, cleaning, transforming, and exporting data. The first module of this course is largely a repeat of what you already saw in the first module of the course you're currently watching. So feel free to just scan through the first module for the key points. If you're interested in learning more about creating descriptive statistics in R, I also recommend watching my course on

Exploratory Data Analysis with R. The third module, titled Calculating Descriptive Statistics, goes more in-depth into how to create and interpret descriptive statistics. If you're interested in learning more about creating data visualizations with R, I have a three-part series in Pluralsight on Creating Data Visualizations in R. Beginning Data Visualization with R covers how to create and interpret standard data visualizations for one or two categorical or numeric variables using the three main plotting systems in R. That is, the base graphic system, lattice, and ggplot2. Multivariate Data Visualization with R covers how to create and interpret data visualizations for three or more variables using the three main plotting systems in R. Mastering Data Visualization with R covers how to create and interpret data visualizations involving spatial data, hierarchical data, graph and network data, and textural data. In addition, it covers how to create animated data visualizations and interactive data visualizations using Shiny. You can find all three of these courses at my author page on the Pluralsight website using the URL below. To learn more about making predictions using machine learning, Jerry Kurata has a two-hour course on understanding machine learning with R. This course will walk you through all of the steps involved in machine learning, including data preparation, algorithm selection, training a model, and testing a model's accuracy. In addition, the last module of my Exploratory Data Analysis with R course will walk you through how to perform unsupervised learning using cluster analysis and supervised learning using linear regression. At the time this course was created, the courses I previously mentioned were all of the in-depth courses on data science with R in the Pluralsight library. However, there are new courses being created at Pluralsight every day; so I recommend performing a search for courses on R and the specific topic you're interested in learning more about. It's quite likely that by the time you're watching this course, a more in-depth course will have been created and published. The Revolutions blog post is a great way to keep up with industry news for R. It contains great information on big data analysis, predictive modeling, and data science with R. The R-Bloggers website is an excellent resource for keeping up with the R community. It contains aggregated blog posts from R experts, tutorials, and links to other learning resources. For more information on data visualization with R, I recommend Nathan Yau's website, Flowing Data. It's a great collection of information and tutorials on cutting edge data visualization, largely using R. Finally R-exercises contains a large collection of links to short tutorials, as well as full online courses for a variety of topics involving data science using R. They even have a course finder that helps you find the courses you need, given your topic of interest, experience level, and budget.

Course Summary

Before we wrap things up for this course, and for the series as a whole, feedback is very important to me. I use your feedback to improve each and every one of my courses, so please be sure to take a moment to rate this course, ask questions in the discussion board if there's something in this course that you didn't understand, or would like me to clarify, leave comments to let me know what you liked about this course or if you think there is something I could to improve upon future courses, and finally, feel free to send me a tweet on Twitter if you like this course and would like to provide me with feedback in public, as well. My Twitter handle is @matthewrenze. In this course, first, we started with an introduction to data science, why data science is important, and how the data science process works. Next, we learned about the R programming language, why R has become so popular, and how to program with R using RStudio. Then, we learned how to work with data, that is, we learned how to import, clean, transform, and export our data. Next, we learned how to create descriptive statistics with R. This included summary statistics for both categorical and numeric variables. Then, we learned how to create data visualizations in R. We created these data visualizations using ggplot2. Next, we learned how to create statistical models using R. We also demonstrated how to create a linear regression model. Then we learned how to handle big data in R. We used two big data extension packages, that is, FF and Big LM to create a linear model for our big data. Next, we learned how to make predictions with machine learning algorithms. We created a decision-tree classifier to predict the species of Iris flower, based on its dimensions. Finally, we learned how to deploy R into production. We demonstrated how to create an interactive, web-based application using Shiny. Finally, I'd like to thank you for joining me for this introduction to Data Science with R. I hope that you've learned some valuable new skills that you'll be able to put to great use, and I hope to see you again in another Pluralsight course in the future.

Course author

Matthew Renze

Matthew is a data science consultant, author, and international public speaker. He has over 17 years of professional experience working with tech startups to Fortune 500 companies. He is a...

Course info

Level	Beginner
Rating	★★★★★ (407)
My rating	★★★★★
Duration	2h 30m
Released	25 Oct 2016

Share course