3

Review of Basic Data Analytic Methods Using R

Key Concepts

Basic features of R
Data exploration and analysis with R
Statistical methods for evaluation

The previous chapter presented the six phases of the Data Analytics Lifecycle.

- Phase 1: Discovery
- Phase 2: Data Preparation
- Phase 3: Model Planning
- Phase 4: Model Building
- Phase 5: Communicate Results
- Phase 6: Operationalize

The first three phases involve various aspects of data exploration. In general, the success of a data analysis project requires a deep understanding of the data. It also requires a toolbox for mining and presenting the data. These activities include the study of the data in terms of basic statistical measures and creation of graphs and plots to visualize and identify relationships and patterns. Several free or commercial tools are available for exploring, conditioning, modeling, and presenting data. Because of its popularity and versatility, the open-source programming language R is used to illustrate many of the presented analytical tasks and models in this book.

This chapter introduces the basic functionality of the R programming language and environment. The first section gives an overview of how to use R to acquire, parse, and filter the data as well as how to obtain some basic descriptive statistics on a dataset. The second section examines using R to perform exploratory data analysis tasks using visualization. The final section focuses on statistical inference, such as hypothesis testing and analysis of variance in R.

3.1 Introduction to R

R is a programming language and software framework for statistical analysis and graphics. Available for use under the GNU General Public License [1], R software and installation instructions can be obtained via the Comprehensive R Archive and Network [2]. This section provides an overview of the basic functionality of R. In later chapters, this foundation in R is utilized to demonstrate many of the presented analytical techniques.

Before delving into specific operations and functions of R later in this chapter, it is important to understand the flow of a basic R script to address an analytical problem. The following R code illustrates a typical analytical situation in which a dataset is imported, the contents of the dataset are examined, and some modeling building tasks are executed. Although the reader may not yet be familiar with the R syntax, the code can be followed by reading the embedded comments, denoted by #. In the following scenario, the annual sales in U.S. dollars for 10,000 retail customers have been provided in the form of a comma-separated-value (CSV) file. The `read.csv()` function is used to import the CSV file. This dataset is stored to the R variable *`sales`* using the assignment operator `<-`.

```
# import a CSV file of the total annual sales for each customer
sales <- read.csv("c:/data/yearly_sales.csv")

# examine the imported dataset
head(sales)
```

```
summary(sales)

# plot num_of_orders vs. sales
plot(sales$num_of_orders,sales$sales_total,
        main="Number of Orders vs. Sales")

# perform a statistical analysis (fit a linear regression model)
results <- lm(sales$sales_total ~ sales$num_of_orders)
summary(results)

# perform some diagnostics on the fitted model
# plot histogram of the residuals
hist(results$residuals, breaks = 800)
```

In this example, the data file is imported using the `read.csv()` function. Once the file has been imported, it is useful to examine the contents to ensure that the data was loaded properly as well as to become familiar with the data. In the example, the `head()` function, by default, displays the first six records of *`sales`*.

```
# examine the imported dataset
head(sales)
  cust_id sales_total num_of_orders gender
1  100001      800.64             3      F
2  100002      217.53             3      F
3  100003       74.58             2      M
4  100004      498.60             3      M
5  100005      723.11             4      F
6  100006       69.43             2      F
```

The `summary()` function provides some descriptive statistics, such as the mean and median, for each data column. Additionally, the minimum and maximum values as well as the 1st and 3rd quartiles are provided. Because the *`gender`* column contains two possible characters, an "F" (female) or "M" (male), the `summary()` function provides the count of each character's occurrence.

```
summary(sales)

    cust_id        sales_total      num_of_orders    gender
 Min.   :100001   Min.   :  30.02   Min.   : 1.000   F:5035
 1st Qu.:102501   1st Qu.:  80.29   1st Qu.: 2.000   M:4965
 Median :105001   Median : 151.65   Median : 2.000
 Mean   :105001   Mean   : 249.46   Mean   : 2.428
 3rd Qu.:107500   3rd Qu.: 295.50   3rd Qu.: 3.000
 Max.   :110000   Max.   :7606.09   Max.   :22.000
```

Plotting a dataset's contents can provide information about the relationships between the various columns. In this example, the `plot()` function generates a scatterplot of the number of orders (`sales$num_of_orders`) against the annual sales (`sales$sales_total`). The `$` is used to reference a specific column in the dataset `sales`. The resulting plot is shown in Figure 3-1.

```
# plot num_of_orders vs. sales
plot(sales$num_of_orders,sales$sales_total,
        main="Number of Orders vs. Sales")
```

FIGURE 3-1 *Graphically examining the data*

Each point corresponds to the number of orders and the total sales for each customer. The plot indicates that the annual sales are proportional to the number of orders placed. Although the observed relationship between these two variables is not purely linear, the analyst decided to apply linear regression using the `lm()` function as a first step in the modeling process.

```
results <- lm(sales$sales_total ~ sales$num_of_orders)
results

Call:
lm(formula = sales$sales_total ~ sales$num_of_orders)

Coefficients:
        (Intercept)  sales$num_of_orders
             -154.1                166.2
```

The resulting intercept and slope values are –154.1 and 166.2, respectively, for the fitted linear equation. However, *`results`* stores considerably more information that can be examined with the `summary()` function. Details on the contents of *`results`* are examined by applying the `attributes()` function. Because regression analysis is presented in more detail later in the book, the reader should not overly focus on interpreting the following output.

```
summary(results)

Call:
lm(formula = sales$sales_total ~ sales$num_of_orders)

Residuals:
   Min     1Q Median     3Q    Max
-666.5 -125.5  -26.7   86.6 4103.4

Coefficients:
                    Estimate Std. Error t value Pr(>|t|)
(Intercept)         -154.128      4.129  -37.33   <2e-16 ***
sales$num_of_orders  166.221      1.462  113.66   <2e-16 ***
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 210.8 on 9998 degrees of freedom
Multiple R-squared:  0.5637,      Adjusted R-squared:  0.5637
F-statistic: 1.292e+04 on 1 and 9998 DF,  p-value: < 2.2e-16
```

The `summary()` function is an example of a generic function. A ***generic function*** is a group of functions sharing the same name but behaving differently depending on the number and the type of arguments they receive. Utilized previously, `plot()` is another example of a generic function; the plot is determined by the passed variables. Generic functions are used throughout this chapter and the book. In the final portion of the example, the following R code uses the generic function `hist()` to generate a histogram (Figure 3-2) of the residuals stored in *`results`*. The function call illustrates that optional parameter values can be passed. In this case, the number of *`breaks`* is specified to observe the large residuals.

```
# perform some diagnostics on the fitted model
# plot histogram of the residuals
hist(results$residuals, breaks = 800)
```

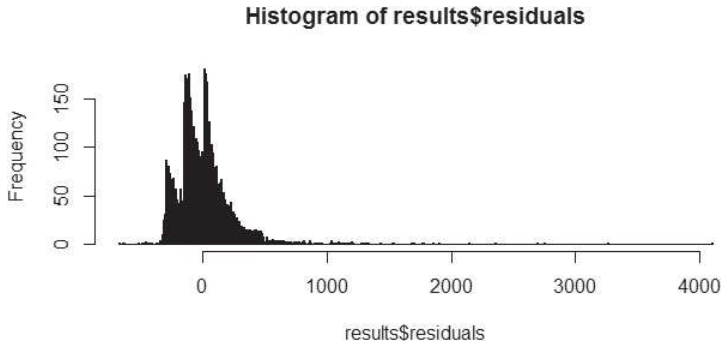

FIGURE 3-2 *Evidence of large residuals*

This simple example illustrates a few of the basic model planning and building tasks that may occur in Phases 3 and 4 of the Data Analytics Lifecycle. Throughout this chapter, it is useful to envision how the presented R functionality will be used in a more comprehensive analysis.

3.1.1 R Graphical User Interfaces

R software uses a command-line interface (CLI) that is similar to the BASH shell in Linux or the interactive versions of scripting languages such as Python. UNIX and Linux users can enter command `R` at the terminal prompt to use the CLI. For Windows installations, R comes with RGui.exe, which provides a basic graphical user interface (GUI). However, to improve the ease of writing, executing, and debugging R code, several additional GUIs have been written for R. Popular GUIs include the R commander [3], Rattle [4], and RStudio [5]. This section presents a brief overview of RStudio, which was used to build the R examples in this book. Figure 3-3 provides a screenshot of the previous R code example executed in RStudio.

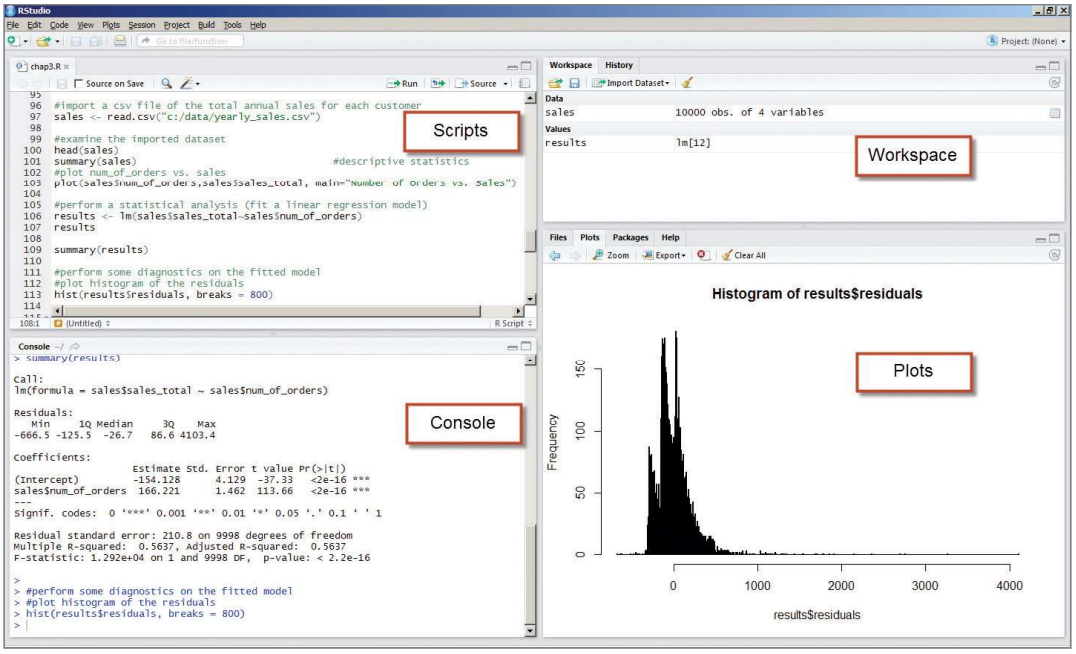

FIGURE 3-3 *RStudio GUI*

The four highlighted window panes follow.

- **Scripts:** Serves as an area to write and save R code
- **Workspace:** Lists the datasets and variables in the R environment
- **Plots:** Displays the plots generated by the R code and provides a straightforward mechanism to export the plots
- **Console:** Provides a history of the executed R code and the output

Additionally, the console pane can be used to obtain help information on R. Figure 3-4 illustrates that by entering `?lm` at the console prompt, the help details of the `lm()` function are provided on the right. Alternatively, `help(lm)` could have been entered at the console prompt.

Functions such as `edit()` and `fix()` allow the user to update the contents of an R variable. Alternatively, such changes can be implemented with RStudio by selecting the appropriate variable from the workspace pane.

R allows one to save the workspace environment, including variables and loaded libraries, into an `.Rdata` file using the `save.image()` function. An existing `.Rdata` file can be loaded using the `load.image()` function. Tools such as RStudio prompt the user for whether the developer wants to save the workspace connects prior to exiting the GUI.

The reader is encouraged to install R and a preferred GUI to try out the R examples provided in the book and utilize the help functionality to access more details about the discussed topics.

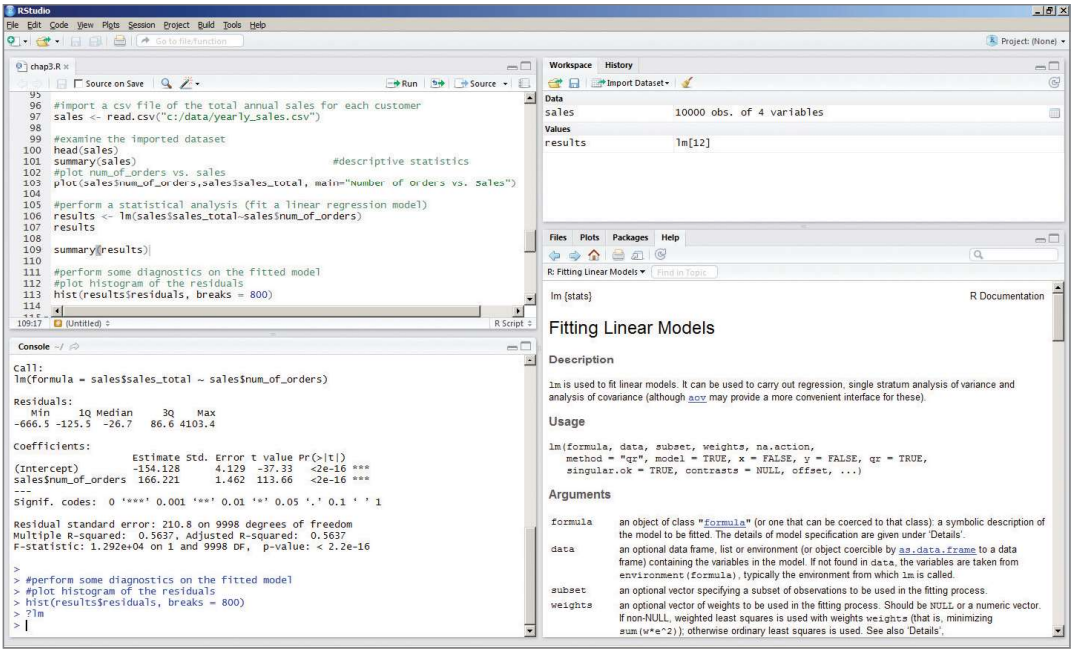

FIGURE 3-4 *Accessing help in Rstudio*

3.1.2 Data Import and Export

In the annual retail sales example, the dataset was imported into R using the `read.csv()` function as in the following code.

```
sales <- read.csv("c:/data/yearly_sales.csv")
```

R uses a forward slash (/) as the separator character in the directory and file paths. This convention makes script files somewhat more portable at the expense of some initial confusion on the part of Windows users, who may be accustomed to using a backslash (\) as a separator. To simplify the import of multiple files with long path names, the `setwd()` function can be used to set the working directory for the subsequent import and export operations, as shown in the following R code.

```
setwd("c:/data/")
sales <- read.csv("yearly_sales.csv")
```

Other import functions include `read.table()` and `read.delim()`, which are intended to import other common file types such as TXT. These functions can also be used to import the `yearly_sales.csv` file, as the following code illustrates.

```
sales_table <- read.table("yearly_sales.csv", header=TRUE, sep=",")
sales_delim <- read.delim("yearly_sales.csv", sep=",")
```

The main difference between these import functions is the default values. For example, the `read.delim()` function expects the column separator to be a tab ("`\t`"). In the event that the numerical data

in a data file uses a comma for the decimal, R also provides two additional functions—`read.csv2()` and `read.delim2()`—to import such data. Table 3-1 includes the expected defaults for headers, column separators, and decimal point notations.

TABLE 3-1 *Import Function Defaults*

Function	Headers	Separator	Decimal Point
`read.table()`	FALSE	" "	"."
`read.csv()`	TRUE	","	"."
`read.csv2()`	TRUE	";"	","
`read.delim()`	TRUE	"\t"	"."
`read.delim2()`	TRUE	"\t"	","

The analogous R functions such as `write.table()`, `write.csv()`, and `write.csv2()` enable exporting of R datasets to an external file. For example, the following R code adds an additional column to the sales dataset and exports the modified dataset to an external file.

```
# add a column for the average sales per order
sales$per_order <- sales$sales_total/sales$num_of_orders

# export data as tab delimited without the row names
write.table(sales,"sales_modified.txt", sep="\t", row.names=FALSE
```

Sometimes it is necessary to read data from a database management system (DBMS). R packages such as `DBI` [6] and `RODBC` [7] are available for this purpose. These packages provide database interfaces for communication between R and DBMSs such as MySQL, Oracle, SQL Server, PostgreSQL, and Pivotal Greenplum. The following R code demonstrates how to install the `RODBC` package with the `install.packages()` function. The `library()` function loads the package into the R workspace. Finally, a connector (*`conn`*) is initialized for connecting to a Pivotal Greenplum database *`training2`* via open database connectivity (ODBC) with user `user`. The *`training2`* database must be defined either in the `/etc/ODBC.ini` configuration file or using the Administrative Tools under the Windows Control Panel.

```
install.packages("RODBC")
library(RODBC)
conn <- odbcConnect("training2", uid="user", pwd="password")
```

The connector needs to be present to submit a SQL query to an ODBC database by using the `sqlQuery()` function from the `RODBC` package. The following R code retrieves specific columns from the *`housing`* table in which household income (`hinc`) is greater than $1,000,000.

```
housing_data <- sqlQuery(conn, "select serialno, state, persons, rooms
                                from housing
                                where hinc > 1000000")
head(housing_data)
  serialno state persons rooms
1  3417867     6       2     7
2  3417867     6       2     7
```

```
3  4552088     6        5     9
4  4552088     6        5     9
5  8699293     6        5     5
6  8699293     6        5     5
```

Although plots can be saved using the RStudio GUI, plots can also be saved using R code by specifying the appropriate graphic devices. Using the `jpeg()` function, the following R code creates a new JPEG file, adds a histogram plot to the file, and then closes the file. Such techniques are useful when automating standard reports. Other functions, such as `png()`, `bmp()`, `pdf()`, and `postscript()`, are available in R to save plots in the desired format.

```
jpeg(file="c:/data/sales_hist.jpeg") # create a new jpeg file
hist(sales$num_of_orders)            # export histogram to jpeg
dev.off()                            # shut off the graphic device
```

More information on data imports and exports can be found at `http://cran.r-project.org/doc/manuals/r-release/R-data.html`, such as how to import datasets from statistical software packages including Minitab, SAS, and SPSS.

3.1.3 Attribute and Data Types

In the earlier example, the *`sales`* variable contained a record for each customer. Several characteristics, such as total annual sales, number of orders, and gender, were provided for each customer. In general, these characteristics or attributes provide the qualitative and quantitative measures for each item or subject of interest. Attributes can be categorized into four types: nominal, ordinal, interval, and ratio (NOIR) [8]. Table 3-2 distinguishes these four attribute types and shows the operations they support. Nominal and ordinal attributes are considered categorical attributes, whereas interval and ratio attributes are considered numeric attributes.

TABLE 3-2 *NOIR Attribute Types*

	Categorical (Qualitative)		Numeric (Quantitative)	
	Nominal	**Ordinal**	**Interval**	**Ratio**
Definition	The values represent labels that distinguish one from another.	Attributes imply a sequence.	The difference between two values is meaningful.	Both the difference and the ratio of two values are meaningful.
Examples	ZIP codes, nationality, street names, gender, employee ID numbers, `TRUE` or `FALSE`	Quality of diamonds, academic grades, magnitude of earthquakes	Temperature in Celsius or Fahrenheit, calendar dates, latitudes	Age, temperature in Kelvin, counts, length, weight
Operations	$=, \neq$	$=, \neq,$ $<, \leq, >, \geq$	$=, \neq,$ $<, \leq, >, \geq,$ $+, -$	$=, \neq,$ $<, \leq, >, \geq,$ $+, -,$ $\times, \div$

Data of one attribute type may be converted to another. For example, the *quality* of diamonds {Fair, Good, Very Good, Premium, Ideal} is considered ordinal but can be converted to nominal {Good, Excellent} with a defined mapping. Similarly, a ratio attribute like *Age* can be converted into an ordinal attribute such as {Infant, Adolescent, Adult, Senior}. Understanding the attribute types in a given dataset is important to ensure that the appropriate descriptive statistics and analytic methods are applied and properly interpreted. For example, the mean and standard deviation of U.S. postal ZIP codes are not very meaningful or appropriate. Proper handling of categorical variables will be addressed in subsequent chapters. Also, it is useful to consider these attribute types during the following discussion on R data types.

Numeric, Character, and Logical Data Types

Like other programming languages, R supports the use of numeric, character, and logical (Boolean) values. Examples of such variables are given in the following R code.

```
i <- 1                          # create a numeric variable
sport <- "football"             # create a character variable
flag <- TRUE                    # create a logical variable
```

R provides several functions, such as `class()` and `typeof()`, to examine the characteristics of a given variable. The `class()` function represents the abstract class of an object. The `typeof()` function determines the way an object is stored in memory. Although *i* appears to be an integer, *i* is internally stored using double precision. To improve the readability of the code segments in this section, the inline R comments are used to explain the code or to provide the returned values.

```
class(i)                        # returns "numeric"
typeof(i)                       # returns "double"

class(sport)                    # returns "character"
typeof(sport)                   # returns "character"

class(flag)                     # returns "logical"
typeof(flag)                    # returns "logical"
```

Additional R functions exist that can test the variables and coerce a variable into a specific type. The following R code illustrates how to test if *i* is an integer using the `is.integer()` function and to coerce *i* into a new integer variable, *j*, using the `as.integer()` function. Similar functions can be applied for double, character, and logical types.

```
is.integer(i)                   # returns FALSE
j <- as.integer(i)              # coerces contents of i into an integer
is.integer(j)                   # returns TRUE
```

The application of the `length()` function reveals that the created variables each have a length of 1. One might have expected the returned length of *sport* to have been 8 for each of the characters in the string `"football"`. However, these three variables are actually one element, ***vectors***.

```
length(i)                       # returns 1
length(flag)                    # returns 1
length(sport)                   # returns 1 (not 8 for "football")
```

Vectors

Vectors are a basic building block for data in R. As seen previously, simple R variables are actually vectors. A vector can only consist of values in the same class. The tests for vectors can be conducted using the `is.vector()` function.

```
is.vector(i)                    # returns TRUE
is.vector(flag)                 # returns TRUE
is.vector(sport)                # returns TRUE
```

R provides functionality that enables the easy creation and manipulation of vectors. The following R code illustrates how a vector can be created using the combine function, `c()` or the colon operator, `:`, to build a vector from the sequence of integers from 1 to 5. Furthermore, the code shows how the values of an existing vector can be easily modified or accessed. The code, related to the *z* vector, indicates how logical comparisons can be built to extract certain elements of a given vector.

```
u <- c("red", "yellow", "blue") # create a vector "red" "yellow" "blue"
u                               # returns "red" "yellow" "blue"
u[1]                            # returns "red" (1st element in u)
v <- 1:5                        # create a vector 1 2 3 4 5
v                               # returns 1 2 3 4 5
sum(v)                          # returns 15
w <- v * 2                      # create a vector 2 4 6 8 10
w                               # returns 2 4 6 8 10
w[3]                            # returns 6 (the 3rd element of w)
z <- v + w                      # sums two vectors element by element
z                               # returns 3  6  9 12 15
z > 8                           # returns FALSE FALSE  TRUE  TRUE  TRUE
z[z > 8]                        # returns 9 12 15
z[z > 8 | z < 5]                # returns 3  9 12 15 ("|" denotes "or")
```

Sometimes it is necessary to initialize a vector of a specific length and then populate the content of the vector later. The `vector()` function, by default, creates a logical vector. A vector of a different type can be specified by using the `mode` parameter. The vector *c*, an integer vector of length 0, may be useful when the number of elements is not initially known and the new elements will later be added to the end of the vector as the values become available.

```
a <- vector(length=3)           # create a logical vector of length 3
a                               # returns FALSE FALSE FALSE
b <- vector(mode="numeric", 3)  # create a numeric vector of length 3
typeof(b)                       # returns "double"
b[2] <- 3.1                     # assign 3.1 to the 2nd element
b                               # returns 0.0 3.1 0.0
c <- vector(mode="integer", 0)  # create an integer vector of length 0
c                               # returns integer(0)
length(c)                       # returns 0
```

Although vectors may appear to be analogous to arrays of one dimension, they are technically dimensionless, as seen in the following R code. The concept of arrays and matrices is addressed in the following discussion.

```
length(b)                          # returns 3
dim(b)                             # returns NULL (an undefined value)
```

Arrays and Matrices

The `array()` function can be used to restructure a vector as an array. For example, the following R code builds a three-dimensional array to hold the quarterly sales for three regions over a two-year period and then assign the sales amount of $158,000 to the second region for the first quarter of the first year.

```
# the dimensions are 3 regions, 4 quarters, and 2 years
quarterly_sales <- array(0, dim=c(3,4,2))
quarterly_sales[2,1,1] <-  158000
quarterly_sales

, , 1

       [,1] [,2] [,3] [,4]
[1,]      0    0    0    0
[2,] 158000    0    0    0
[3,]      0    0    0    0

, , 2

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
```

A two-dimensional array is known as a ***matrix***. The following code initializes a matrix to hold the quarterly sales for the three regions. The parameters `nrow` and `ncol` define the number of rows and columns, respectively, for the *`sales_matrix`*.

```
sales_matrix <- matrix(0, nrow = 3, ncol = 4)
sales_matrix

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
```

R provides the standard matrix operations such as addition, subtraction, and multiplication, as well as the transpose function `t()` and the inverse matrix function `matrix.inverse()` included in the `matrixcalc` package. The following R code builds a 3×3 matrix, M, and multiplies it by its inverse to obtain the identity matrix.

```
library(matrixcalc)
M <- matrix(c(1,3,3,5,0,4,3,3,3),nrow = 3,ncol = 3) # build a 3x3 matrix
```

```
M %*% matrix.inverse(M)                    # multiply M by inverse(M)

     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Data Frames

Similar to the concept of matrices, data frames provide a structure for storing and accessing several variables of possibly different data types. In fact, as the `is.data.frame()` function indicates, a data frame was created by the `read.csv()` function at the beginning of the chapter.

```
#import a CSV file of the total annual sales for each customer
sales <- read.csv("c:/data/yearly_sales.csv")
is.data.frame(sales)            # returns TRUE
```

As seen earlier, the variables stored in the data frame can be easily accessed using the `$` notation. The following R code illustrates that in this example, each variable is a vector with the exception of `gender`, which was, by a `read.csv()` default, imported as a ***factor***. Discussed in detail later in this section, a factor denotes a categorical variable, typically with a few finite levels such as "F" and "M" in the case of *`gender`*.

```
length(sales$num_of_orders)     # returns 10000 (number of customers)

is.vector(sales$cust_id)        # returns TRUE
is.vector(sales$sales_total)    # returns TRUE
is.vector(sales$num_of_orders)  # returns TRUE
is.vector(sales$gender)         # returns FALSE

is.factor(sales$gender)         # returns TRUE
```

Because of their flexibility to handle many data types, data frames are the preferred input format for many of the modeling functions available in R. The following use of the `str()` function provides the structure of the *`sales`* data frame. This function identifies the integer and numeric (double) data types, the factor variables and levels, as well as the first few values for each variable.

```
str(sales)              # display structure of the data frame object

'data.frame': 10000 obs. of 4 variables:
 $ cust_id      : int  100001 100002 100003 100004 100005 100006 ...
 $ sales_total  : num  800.6 217.5 74.6 498.6 723.1 ...
 $ num_of_orders: int  3 3 2 3 4 2 2 2 2 2 ...
 $ gender       : Factor w/ 2 levels "F","M": 1 1 2 2 1 1 2 2 1 2 ...
```

In the simplest sense, data frames are lists of variables of the same length. A subset of the data frame can be retrieved through ***subsetting operators***. R's subsetting operators are powerful in that they allow one to express complex operations in a succinct fashion and easily retrieve a subset of the dataset.

```
# extract the fourth column of the sales data frame
sales[,4]
# extract the gender column of the sales data frame
```

```
sales$gender
# retrieve the first two rows of the data frame
sales[1:2,]
# retrieve the first, third, and fourth columns
sales[,c(1,3,4)]
# retrieve both the cust_id and the sales_total columns
sales[,c("cust_id", "sales_total")]
# retrieve all the records whose gender is female
sales[sales$gender=="F",]
```

The following R code shows that the class of the `sales` variable is a data frame. However, the type of the `sales` variable is a list. A ***list*** is a collection of objects that can be of various types, including other lists.

```
class(sales)
"data.frame"
typeof(sales)
"list"
```

Lists

Lists can contain any type of objects, including other lists. Using the vector v and the matrix M created in earlier examples, the following R code creates *`assortment`*, a list of different object types.

```
# build an assorted list of a string, a numeric, a list, a vector,
# and a matrix
housing <- list("own", "rent")
assortment <- list("football", 7.5, housing, v, M)
assortment

[[1]]
[1] "football"

[[2]]
[1] 7.5

[[3]]
[[3]][[1]]
[1] "own"

[[3]][[2]]
[1] "rent"


[[4]]
[1] 1 2 3 4 5

[[5]]
```

```
     [,1] [,2] [,3]
[1,]    1    5    3
[2,]    3    0    3
[3,]    3    4    3
```

In displaying the contents of *assortment*, the use of the double brackets, `[[]]`, is of particular importance. As the following R code illustrates, the use of the single set of brackets only accesses an item in the list, not its content.

```
# examine the fifth object, M, in the list
class(assortment[5])            # returns "list"
length(assortment[5])           # returns 1

class(assortment[[5]])          # returns "matrix"
length(assortment[[5]])         # returns 9 (for the 3x3 matrix)
```

As presented earlier in the data frame discussion, the `str()` function offers details about the structure of a list.

```
str(assortment)
List of 5
 $ : chr "football"
 $ : num 7.5
 $ :List of 2
  ..$ : chr "own"
  ..$ : chr "rent"
 $ : int [1:5] 1 2 3 4 5
 $ : num [1:3, 1:3] 1 3 3 5 0 4 3 3 3
```

Factors

Factors were briefly introduced during the discussion of the *gender* variable in the data frame *sales*. In this case, *gender* could assume one of two levels: `F` or `M`. Factors can be ordered or not ordered. In the case of *gender*, the levels are not ordered.

```
class(sales$gender)             # returns "factor"
is.ordered(sales$gender)        # returns FALSE
```

Included with the `ggplot2` package, the *diamonds* data frame contains three ordered factors. Examining the *cut* factor, there are five levels in order of improving cut: Fair, Good, Very Good, Premium, and Ideal. Thus, `sales$gender` contains nominal data, and `diamonds$cut` contains ordinal data.

```
head(sales$gender)    # display first six values and the levels

F F M M F F
Levels: F M

library(ggplot2)
data(diamonds)         # load the data frame into the R workspace
```

```
str(diamonds)
'data.frame': 53940 obs. of 10 variables:
 $ carat  : num  0.23 0.21 0.23 0.29 0.31 0.24 0.24 0.26 0.22 ...
 $ cut    : Ord.factor w/ 5 levels "Fair"<"Good"<..: 5 4 2 4 2 3 ...
 $ color  : Ord.factor w/ 7 levels "D"<"E"<"F"<"G"<..: 2 2 2 6 7 7 ...
 $ clarity: Ord.factor w/ 8 levels "I1"<"SI2"<"SI1"<..: 2 3 5 4 2  ...
 $ depth  : num  61.5 59.8 56.9 62.4 63.3 62.8 62.3 61.9 65.1 59.4 ...
 $ table  : num  55 61 65 58 58 57 57 55 61 61 ...
 $ price  : int  326 326 327 334 335 336 336 337 337 338 ...
 $ x      : num  3.95 3.89 4.05 4.2 4.34 3.94 3.95 4.07 3.87 4 ...
 $ y      : num  3.98 3.84 4.07 4.23 4.35 3.96 3.98 4.11 3.78 4.05 ...
 $ z      : num  2.43 2.31 2.31 2.63 2.75 2.48 2.47 2.53 2.49 2.39 ...

head(diamonds$cut)       # display first six values and the levels
Ideal     Premium   Good      Premium   Good      Very Good
Levels: Fair < Good < Very Good < Premium < Ideal
```

Suppose it is decided to categorize *sales$sales_totals* into three groups—small, medium, and big—according to the amount of the sales with the following code. These groupings are the basis for the new ordinal factor, spender, with levels {small, medium, big}.

```
# build an empty character vector of the same length as sales
sales_group <- vector(mode="character",
                      length=length(sales$sales_total))

# group the customers according to the sales amount
sales_group[sales$sales_total<100] <- "small"
sales_group[sales$sales_total>=100 & sales$sales_total<500] <- "medium"
sales_group[sales$sales_total>=500] <- "big"

# create and add the ordered factor to the sales data frame
spender <- factor(sales_group,levels=c("small", "medium", "big"),
                              ordered = TRUE)
sales <- cbind(sales,spender)

str(sales$spender)
Ord.factor w/ 3 levels "small"<"medium"<..: 3 2 1 2 3 1 1 1 2 1 ...

head(sales$spender)
big    medium small  medium big    small
Levels: small < medium < big
```

The `cbind()` function is used to combine variables column-wise. The `rbind()` function is used to combine datasets row-wise. The use of factors is important in several R statistical modeling functions, such as analysis of variance, `aov()`, presented later in this chapter, and the use of contingency tables, discussed next.

Contingency Tables

In R, ***table*** refers to a class of objects used to store the observed counts across the factors for a given dataset. Such a table is commonly referred to as a contingency table and is the basis for performing a statistical test on the independence of the factors used to build the table. The following R code builds a contingency table based on the `sales$gender` and `sales$spender` factors.

```
# build a contingency table based on the gender and spender factors
sales_table <- table(sales$gender,sales$spender)
sales_table
  small medium  big
F  1726   2746  563
M  1656   2723  586

class(sales_table)              # returns "table"
typeof(sales_table)             # returns "integer"
dim(sales_table)                # returns 2 3

# performs a chi-squared test
summary(sales_table)
Number of cases in table: 10000
Number of factors: 2
Test for independence of all factors:
  Chisq = 1.516, df = 2, p-value = 0.4686
```

Based on the observed counts in the table, the `summary()` function performs a chi-squared test on the independence of the two factors. Because the reported *p*-value is greater than 0.05, the assumed independence of the two factors is not rejected. Hypothesis testing and *p*-values are covered in more detail later in this chapter. Next, applying descriptive statistics in R is examined.

3.1.4 Descriptive Statistics

It has already been shown that the `summary()` function provides several descriptive statistics, such as the mean and median, about a variable such as the *`sales`* data frame. The results now include the counts for the three levels of the *`spender`* variable based on the earlier examples involving factors.

```
summary(sales)
    cust_id        sales_total      num_of_orders    gender     spender
 Min.   :100001   Min.   :  30.02   Min.   : 1.000  F:5035  small :3382
 1st Qu.:102501   1st Qu.:  80.29   1st Qu.: 2.000  M:4965  medium:5469
 Median :105001   Median : 151.65   Median : 2.000          big   :1149
 Mean   :105001   Mean   : 249.46   Mean   : 2.428
 3rd Qu.:107500   3rd Qu.: 295.50   3rd Qu.: 3.000
 Max.   :110000   Max.   :7606.09   Max.   :22.000
```

The following code provides some common R functions that include descriptive statistics. In parentheses, the comments describe the functions.

```
# to simplify the function calls, assign
x <- sales$sales_total
y <- sales$num_of_orders

cor(x,y)                          # returns 0.7508015 (correlation)
cov(x,y)                          # returns 345.2111 (covariance)
IQR(x)                            # returns 215.21 (interquartile range)
mean(x)                           # returns 249.4557 (mean)
median(x)                         # returns 151.65 (median)
range(x)                          # returns 30.02 7606.09 (min max)
sd(x)                             # returns 319.0508 (std. dev.)
var(x)                            # returns 101793.4 (variance)
```

The `IQR()` function provides the difference between the third and the first quartiles. The other functions are fairly self-explanatory by their names. The reader is encouraged to review the available help files for acceptable inputs and possible options.

The function `apply()` is useful when the same function is to be applied to several variables in a data frame. For example, the following R code calculates the standard deviation for the first three variables in *`sales`*. In the code, setting `MARGIN=2` specifies that the `sd()` function is applied over the columns. Other functions, such as `lapply()` and `sapply()`, apply a function to a list or vector. Readers can refer to the R help files to learn how to use these functions.

```
apply(sales[,c(1:3)], MARGIN=2, FUN=sd)
      cust_id   sales_total num_of_orders
  2886.895680    319.050782      1.441119
```

Additional descriptive statistics can be applied with user-defined functions. The following R code defines a function, `my_range()`, to compute the difference between the maximum and minimum values returned by the `range()` function. In general, user-defined functions are useful for any task or operation that needs to be frequently repeated. More information on user-defined functions is available by entering `help("function")` in the console.

```
# build a function to provide the difference between
# the maximum and the minimum values
my_range <- function(v) {range(v)[2] - range(v)[1]}
my_range(x)
7576.07
```

3.2 Exploratory Data Analysis

So far, this chapter has addressed importing and exporting data in R, basic data types and operations, and generating descriptive statistics. Functions such as `summary()` can help analysts easily get an idea of the magnitude and range of the data, but other aspects such as linear relationships and distributions are more difficult to see from descriptive statistics. For example, the following code shows a summary view of a data frame *`data`* with two columns *x* and *y*. The output shows the range of *x* and *y*, but it's not clear what the relationship may be between these two variables.

```
summary(data)
       x                   y
 Min.   :-1.90483   Min.   :-2.16545
 1st Qu.:-0.66321   1st Qu.:-0.71451
 Median : 0.09367   Median :-0.03797
 Mean   : 0.02522   Mean   :-0.02153
 3rd Qu.: 0.65414   3rd Qu.: 0.55738
 Max.   : 2.18471   Max.   : 1.70199
```

A useful way to detect patterns and anomalies in the data is through the exploratory data analysis with visualization. Visualization gives a succinct, holistic view of the data that may be difficult to grasp from the numbers and summaries alone. Variables `x` and `y` of the data frame `data` can instead be visualized in a scatterplot (Figure 3-5), which easily depicts the relationship between two variables. An important facet of the initial data exploration, visualization assesses data cleanliness and suggests potentially important relationships in the data prior to the model planning and building phases.

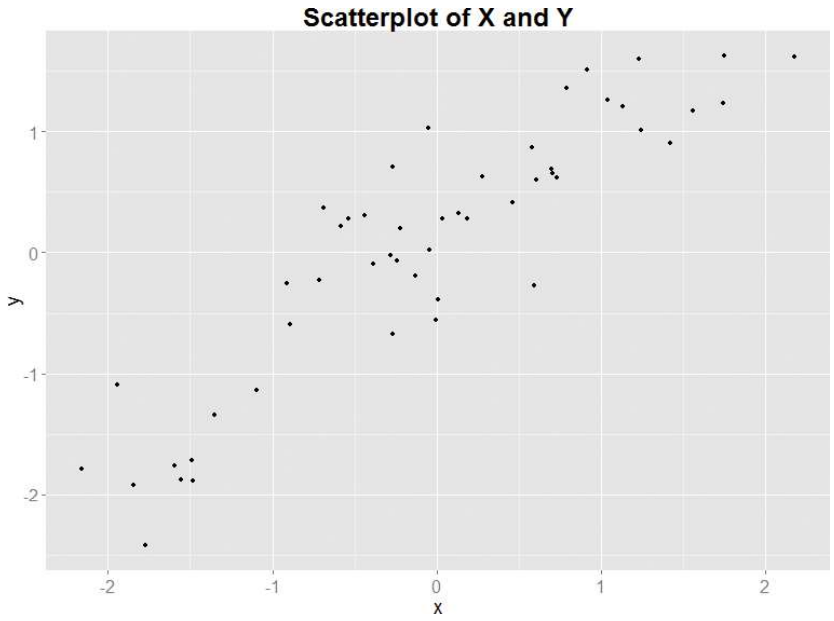

FIGURE 3-5 *A scatterplot can easily show if x and y share a relation*

The code to generate `data` as well as Figure 3-5 is shown next.

```
x <- rnorm(50)
y <- x + rnorm(50, mean=0, sd=0.5)

data <- as.data.frame(cbind(x, y))
```

```
summary(data)

library(ggplot2)
ggplot(data, aes(x=x, y=y)) +
  geom_point(size=2) +
  ggtitle("Scatterplot of X and Y") +
  theme(axis.text=element_text(size=12),
        axis.title = element_text(size=14),
        plot.title = element_text(size=20, face="bold"))
```

Exploratory data analysis [9] is a data analysis approach to reveal the important characteristics of a dataset, mainly through visualization. This section discusses how to use some basic visualization techniques and the plotting feature in R to perform exploratory data analysis.

3.2.1 Visualization Before Analysis

To illustrate the importance of visualizing data, consider Anscombe's quartet. Anscombe's quartet consists of four datasets, as shown in Figure 3-6. It was constructed by statistician Francis Anscombe [10] in 1973 to demonstrate the importance of graphs in statistical analyses.

# 1		# 2		# 3		# 4	
x	y	x	y	x	y	x	y
4	4.26	4	3.10	4	5.39	8	5.25
5	5.68	5	4.74	5	5.73	8	5.56
6	7.24	6	6.13	6	6.08	8	5.76
7	4.82	7	7.26	7	6.42	8	6.58
8	6.95	8	8.14	8	6.77	8	6.89
9	8.81	9	8.77	9	7.11	8	7.04
10	8.04	10	9.14	10	7.46	8	7.71
11	8.33	11	9.26	11	7.81	8	7.91
12	10.84	12	9.13	12	8.15	8	8.47
13	7.58	13	8.74	13	12.74	8	8.84
14	9.96	14	8.10	14	8.84	19	12.50

FIGURE 3-6 *Anscombe's quartet*

The four datasets in Anscombe's quartet have nearly identical statistical properties, as shown in Table 3-3.

TABLE 3-3 *Statistical Properties of Anscombe's Quartet*

Statistical Property	Value
Mean of *x*	9
Variance of *y*	11
Mean of *y*	7.50 (to 2 decimal points)

Variance of *y*	4.12 or 4.13 (to 2 decimal points)
Correlations between *x* and *y*	0.816
Linear regression line	$y = 3.00 + 0.50x$ (to 2 decimal points)

Based on the nearly identical statistical properties across each dataset, one might conclude that these four datasets are quite similar. However, the scatterplots in Figure 3-7 tell a different story. Each dataset is plotted as a scatterplot, and the fitted lines are the result of applying linear regression models. The estimated regression line fits Dataset 1 reasonably well. Dataset 2 is definitely nonlinear. Dataset 3 exhibits a linear trend, with one apparent outlier at $x = 13$. For Dataset 4, the regression line fits the dataset quite well. However, with only points at two x values, it is not possible to determine that the linearity assumption is proper.

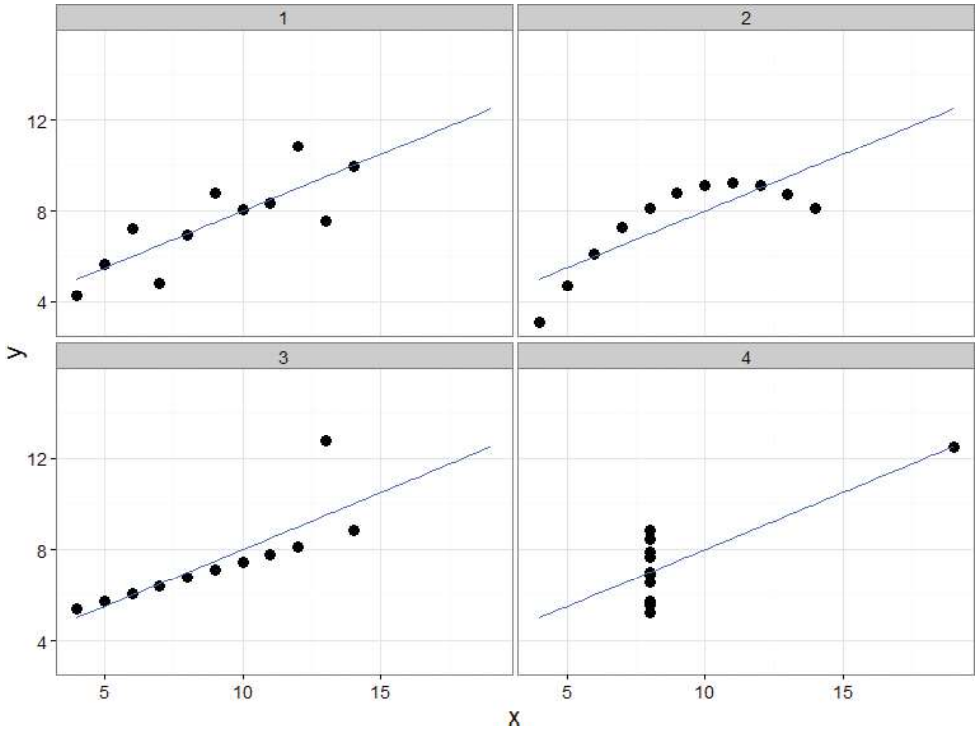

FIGURE 3-7 *Anscombe's quartet visualized as scatterplots*

The R code for generating Figure 3-7 is shown next. It requires the R package `ggplot2` [11], which can be installed simply by running the command `install.packages("ggplot2")`. The `anscombe`

dataset for the plot is included in the standard R distribution. Enter `data()` for a list of datasets included in the R base distribution. Enter `data(DatasetName)` to make a dataset available in the current workspace.

In the code that follows, variable *levels* is created using the `gl()` function, which generates factors of four levels (1, 2, 3, and 4), each repeating 11 times. Variable *mydata* is created using the `with(data, expression)` function, which evaluates an ***expression*** in an environment constructed from ***data***. In this example, the ***data*** is the `anscombe` dataset, which includes eight attributes: *x1*, *x2*, *x3*, *x4*, *y1*, *y2*, *y3*, and *y4*. The ***expression*** part in the code creates a data frame from the `anscombe` dataset, and it only includes three attributes: *x*, *y*, and the group each data point belongs to (*mygroup*).

```
install.packages("ggplot2") # not required if package has been installed

data(anscombe)    # load the anscombe dataset into the current workspace
anscombe
   x1 x2 x3 x4    y1   y2    y3    y4
1  10 10 10  8  8.04 9.14  7.46  6.58
2   8  8  8  8  6.95 8.14  6.77  5.76
3  13 13 13  8  7.58 8.74 12.74  7.71
4   9  9  9  8  8.81 8.77  7.11  8.84
5  11 11 11  8  8.33 9.26  7.81  8.47
6  14 14 14  8  9.96 8.10  8.84  7.04
7   6  6  6  8  7.24 6.13  6.08  5.25
8   4  4  4 19  4.26 3.10  5.39 12.50
9  12 12 12  8 10.84 9.13  8.15  5.56
10  7  7  7  8  4.82 7.26  6.42  7.91
11  5  5  5  8  5.68 4.74  5.73  6.89

nrow(anscombe)    # number of rows
[1] 11

# generates levels to indicate which group each data point belongs to
levels <- gl(4, nrow(anscombe))
levels
[1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3
[34] 4 4 4 4 4 4 4 4 4 4 4
Levels: 1 2 3 4

# Group anscombe into a data frame
mydata <- with(anscombe, data.frame(x=c(x1,x2,x3,x4), y=c(y1,y2,y3,y4),
               mygroup=levels))

mydata
    x     y mygroup
1  10  8.04       1
2   8  6.95       1
3  13  7.58       1
4   9  8.81       1
...
```

```
41 19 12.50        4
42  8  5.56        4
43  8  7.91        4
44  8  6.89        4

# Make scatterplots using the ggplot2 package
library(ggplot2)
theme_set(theme_bw())    # set plot color theme

# create the four plots of Figure 3-7
ggplot(mydata, aes(x,y)) +
      geom_point(size=4) +
      geom_smooth(method="lm", fill=NA, fullrange=TRUE) +
      facet_wrap(~mygroup)
```

3.2.2 Dirty Data

This section addresses how dirty data can be detected in the data exploration phase with visualizations. In general, analysts should look for anomalies, verify the data with domain knowledge, and decide the most appropriate approach to clean the data.

Consider a scenario in which a bank is conducting data analyses of its account holders to gauge customer retention. Figure 3-8 shows the age distribution of the account holders.

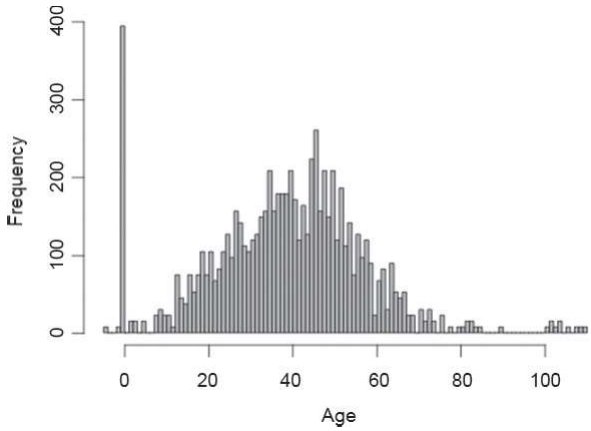

FIGURE 3-8 *Age distribution of bank account holders*

If the age data is in a vector called `age`, the graph can be created with the following R script:

```
hist(age, breaks=100, main="Age Distribution of Account Holders",
     xlab="Age", ylab="Frequency", col="gray")
```

The figure shows that the median age of the account holders is around 40. A few accounts with account holder age less than 10 are unusual but plausible. These could be custodial accounts or college savings accounts set up by the parents of young children. These accounts should be retained for future analyses.

However, the left side of the graph shows a huge spike of customers who are zero years old or have negative ages. This is likely to be evidence of ***missing data***. One possible explanation is that the null age values could have been replaced by 0 or negative values during the data input. Such an occurrence may be caused by entering age in a text box that only allows numbers and does not accept empty values. Or it might be caused by transferring data among several systems that have different definitions for null values (such as NULL, NA, 0, –1, or –2). Therefore, ***data cleansing*** needs to be performed over the accounts with abnormal age values. Analysts should take a closer look at the records to decide if the missing data should be eliminated or if an appropriate age value can be determined using other available information for each of the accounts.

In R, the `is.na()` function provides tests for missing values. The following example creates a vector *x* where the fourth value is not available (`NA`). The `is.na()` function returns `TRUE` at each `NA` value and `FALSE` otherwise.

```
x <- c(1, 2, 3, NA, 4)
is.na(x)
[1] FALSE FALSE FALSE TRUE FALSE
```

Some arithmetic functions, such as `mean()`, applied to data containing missing values can yield an `NA` result. To prevent this, set the `na.rm` parameter to `TRUE` to remove the missing value during the function's execution.

```
mean(x)
[1] NA
mean(x, na.rm=TRUE)
[1] 2.5
```

The `na.exclude()` function returns the object with incomplete cases removed.

```
DF <- data.frame(x = c(1, 2, 3), y = c(10, 20, NA))
DF
  x  y
1 1 10
2 2 20
3 3 NA

DF1 <- na.exclude(DF)
DF1
  x  y
1 1 10
2 2 20
```

Account holders older than 100 may be due to bad data caused by typos. Another possibility is that these accounts may have been passed down to the heirs of the original account holders without being updated. In this case, one needs to further examine the data and conduct data cleansing if necessary. The dirty data could be simply removed or filtered out with an age threshold for future analyses. If removing records is not an option, the analysts can look for patterns within the data and develop a set of heuristics to attack the problem of dirty data. For example, wrong age values could be replaced with ***approximation*** based on the nearest neighbor—the record that is the most similar to the record in question based on analyzing the differences in all the other variables besides age.

Figure 3-9 presents another example of dirty data. The distribution shown here corresponds to the age of mortgages in a bank's home loan portfolio. The mortgage age is calculated by subtracting the origination date of the loan from the current date. The vertical axis corresponds to the number of mortgages at each mortgage age.

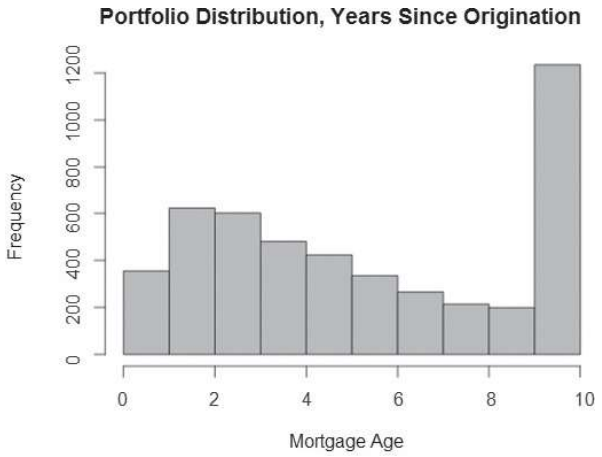

FIGURE 3-9 *Distribution of mortgage in years since origination from a bank's home loan portfolio*

If the data is in a vector called `mortgage`, Figure 3-9 can be produced by the following R script.

```
hist(mortgage, breaks=10, xlab="Mortgage Age", col="gray",
     main="Portfolio Distribution, Years Since Origination")
```

Figure 3-9 shows that the loans are no more than 10 years old, and these 10-year-old loans have a disproportionate frequency compared to the rest of the population. One possible explanation is that the 10-year-old loans do not only include loans originated 10 years ago, but also those originated earlier than that. In other words, the 10 in the *x*-axis actually means ≥ 10. This sometimes happens when data is ported from one system to another or because the data provider decided, for some reason, not to distinguish loans that are more than 10 years old. Analysts need to study the data further and decide the most appropriate way to perform data cleansing.

Data analysts should perform sanity checks against domain knowledge and decide if the dirty data needs to be eliminated. Consider the task to find out the probability of mortgage loan default. If the past observations suggest that most defaults occur before about the 4th year and 10-year-old mortgages rarely default, it may be safe to eliminate the dirty data and assume that the defaulted loans are less than 10 years old. For other analyses, it may become necessary to track down the source and find out the true origination dates.

Dirty data can occur due to acts of omission. In the `sales` data used at the beginning of this chapter, it was seen that the minimum number of orders was 1 and the minimum annual sales amount was $30.02. Thus, there is a strong possibility that the provided dataset did not include the sales data on all customers, just the customers who purchased something during the past year.

3.2.3 Visualizing a Single Variable

Using visual representations of data is a hallmark of exploratory data analyses: letting the data speak to its audience rather than imposing an interpretation on the data *a priori*. Sections 3.2.3 and 3.2.4 examine ways of displaying data to help explain the underlying distributions of a single variable or the relationships of two or more variables.

R has many functions available to examine a single variable. Some of these functions are listed in Table 3-4.

TABLE 3-4 *Example Functions for Visualizing a Single Variable*

Function	Purpose
`plot(data)`	Scatterplot where x is the index and y is the value; suitable for low-volume data
`barplot(data)`	Barplot with vertical or horizontal bars
`dotchart(data)`	Cleveland dot plot [12]
`hist(data)`	Histogram
`plot(density(data))`	Density plot (a continuous histogram)
`stem(data)`	Stem-and-leaf plot
`rug(data)`	Add a rug representation (1-d plot) of the data to an existing plot

Dotchart and Barplot

Dotchart and barplot portray continuous values with labels from a discrete variable. A dotchart can be created in R with the function `dotchart(x, label=...)`, where *x* is a numeric vector and *label* is a vector of categorical labels for *x*. A barplot can be created with the `barplot(height)` function, where *height* represents a vector or matrix. Figure 3-10 shows (a) a dotchart and (b) a barplot based on the `mtcars` dataset, which includes the fuel consumption and 10 aspects of automobile design and performance of 32 automobiles. This dataset comes with the standard R distribution.

The plots in Figure 3-10 can be produced with the following R code.

```
data(mtcars)
dotchart(mtcars$mpg,labels=row.names(mtcars),cex=.7,
         main="Miles Per Gallon (MPG) of Car Models",
         xlab="MPG")
barplot(table(mtcars$cyl), main="Distribution of Car Cylinder Counts",
          xlab="Number of Cylinders")
```

Histogram and Density Plot

Figure 3-11(a) includes a histogram of household income. The histogram shows a clear concentration of low household incomes on the left and the long tail of the higher incomes on the right.

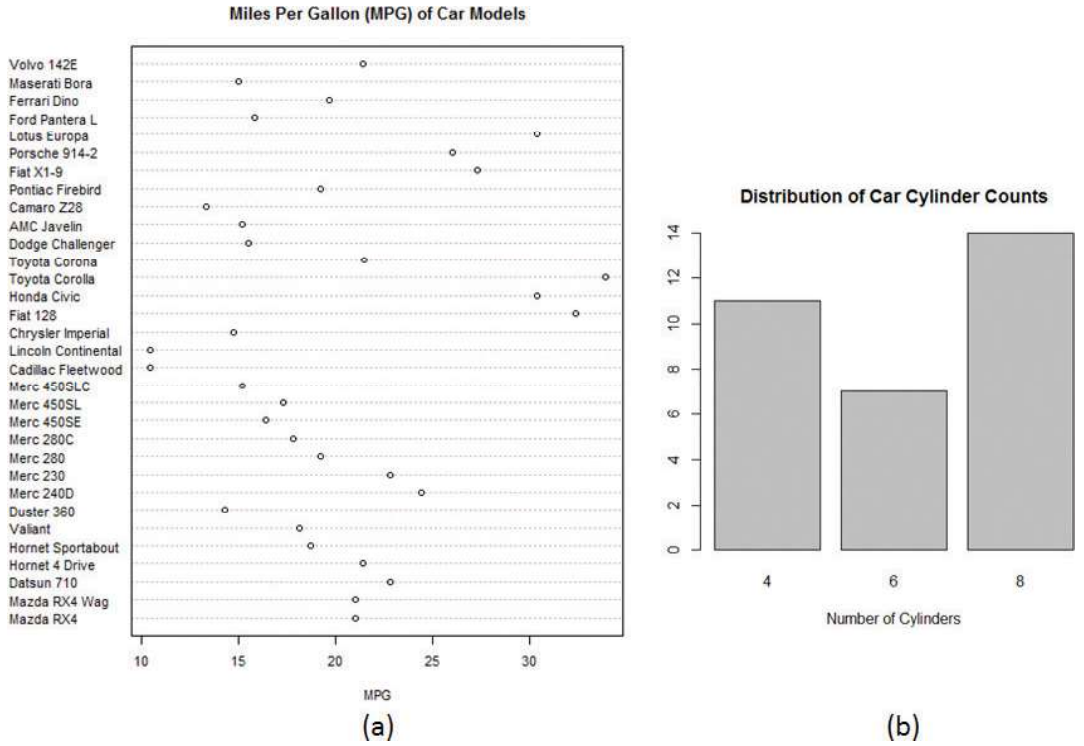

FIGURE 3-10 *(a) Dotchart on the miles per gallon of cars and (b) Barplot on the distribution of car cylinder counts*

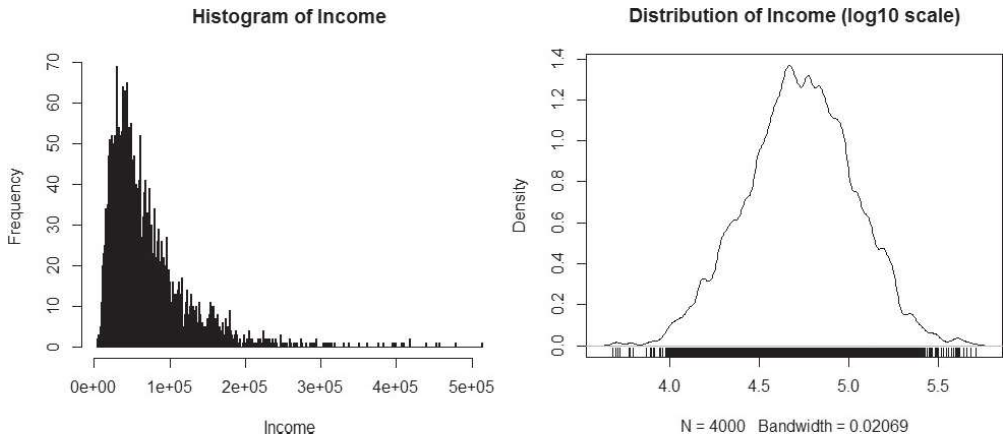

FIGURE 3-11 *(a) Histogram and (b) Density plot of household income*

Figure 3-11(b) shows a density plot of the logarithm of household income values, which emphasizes the distribution. The income distribution is concentrated in the center portion of the graph. The code to generate the two plots in Figure 3-11 is provided next. The `rug()` function creates a one-dimensional density plot on the bottom of the graph to emphasize the distribution of the observation.

```
# randomly generate 4000 observations from the log normal distribution
income <- rlnorm(4000, meanlog = 4, sdlog = 0.7)
summary(income)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.301  33.720  54.970  70.320  88.800 659.800
income <- 1000*income
summary(income)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4301   33720   54970   70320   88800  659800
# plot the histogram
hist(income, breaks=500, xlab="Income", main="Histogram of Income")
# density plot
plot(density(log10(income), adjust=0.5),
     main="Distribution of Income (log10 scale)")
# add rug to the density plot
rug(log10(income))
```

In the data preparation phase of the Data Analytics Lifecycle, the data range and distribution can be obtained. If the data is skewed, viewing the logarithm of the data (if it's all positive) can help detect structures that might otherwise be overlooked in a graph with a regular, nonlogarithmic scale.

When preparing the data, one should look for signs of dirty data, as explained in the previous section. Examining if the data is unimodal or multimodal will give an idea of how many distinct populations with different behavior patterns might be mixed into the overall population. Many modeling techniques assume that the data follows a normal distribution. Therefore, it is important to know if the available dataset can match that assumption before applying any of those modeling techniques.

Consider a density plot of diamond prices (in USD). Figure 3-12(a) contains two density plots for premium and ideal cuts of diamonds. The group of premium cuts is shown in red, and the group of ideal cuts is shown in blue. The range of diamond prices is wide—in this case ranging from around $300 to almost $20,000. Extreme values are typical of monetary data such as income, customer value, tax liabilities, and bank account sizes.

Figure 3-12(b) shows more detail of the diamond prices than Figure 3-12(a) by taking the logarithm. The two humps in the premium cut represent two distinct groups of diamond prices: One group centers around $\log_{10}$ price = 2.9 (where the price is about $794), and the other centers around $\log_{10}$ price = 3.7 (where the price is about $5,012). The ideal cut contains three humps, centering around 2.9, 3.3, and 3.7 respectively.

The R script to generate the plots in Figure 3-12 is shown next. The *diamonds* dataset comes with the `ggplot2` package.

```
library("ggplot2")
data(diamonds)    # load the diamonds dataset from ggplot2

# Only keep the premium and ideal cuts of diamonds
```

```
niceDiamonds <- diamonds[diamonds$cut=="Premium" |
                         diamonds$cut=="Ideal",]

summary(niceDiamonds$cut)
     Fair      Good Very Good   Premium     Ideal
        0         0         0     13791     21551

# plot density plot of diamond prices
ggplot(niceDiamonds, aes(x=price, fill=cut)) +
  geom_density(alpha = .3, color=NA)

# plot density plot of the log10 of diamond prices
ggplot(niceDiamonds, aes(x=log10(price), fill=cut)) +
  geom_density(alpha = .3, color=NA)
```

As an alternative to `ggplot2`, the `lattice` package provides a function called `densityplot()` for making simple density plots.

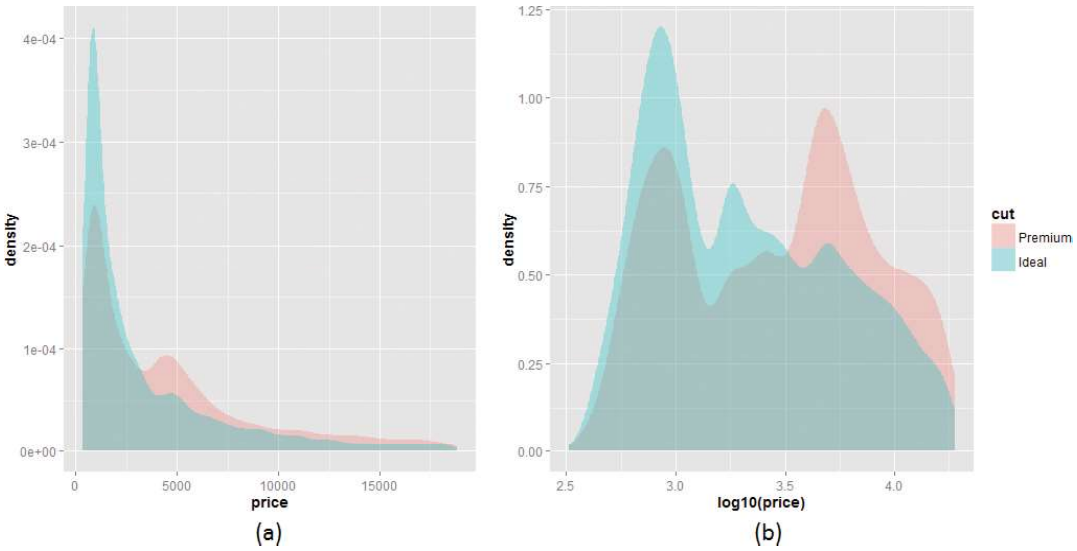

FIGURE 3-12 *Density plots of (a) diamond prices and (b) the logarithm of diamond prices*

3.2.4 Examining Multiple Variables

A scatterplot (shown previously in Figure 3-1 and Figure 3-5) is a simple and widely used visualization for finding the relationship among multiple variables. A scatterplot can represent data with up to five variables using x-axis, y-axis, size, color, and shape. But usually only two to four variables are portrayed in a scatterplot to minimize confusion. When examining a scatterplot, one needs to pay close attention

to the possible relationship between the variables. If the functional relationship between the variables is somewhat pronounced, the data may roughly lie along a straight line, a parabola, or an exponential curve. If variable y is related exponentially to x, then the plot of x versus `log(y)` is approximately linear. If the plot looks more like a cluster without a pattern, the corresponding variables may have a weak relationship.

The scatterplot in Figure 3-13 portrays the relationship of two variables: x and y. The red line shown on the graph is the fitted line from the linear regression. Linear regression will be revisited in Chapter 6, "Advanced Analytical Theory and Methods: Regression." Figure 3-13 shows that the regression line does not fit the data well. This is a case in which linear regression cannot model the relationship between the variables. Alternative methods such as the `loess()` function can be used to fit a nonlinear line to the data. The blue curve shown on the graph represents the LOESS curve, which fits the data better than linear regression.

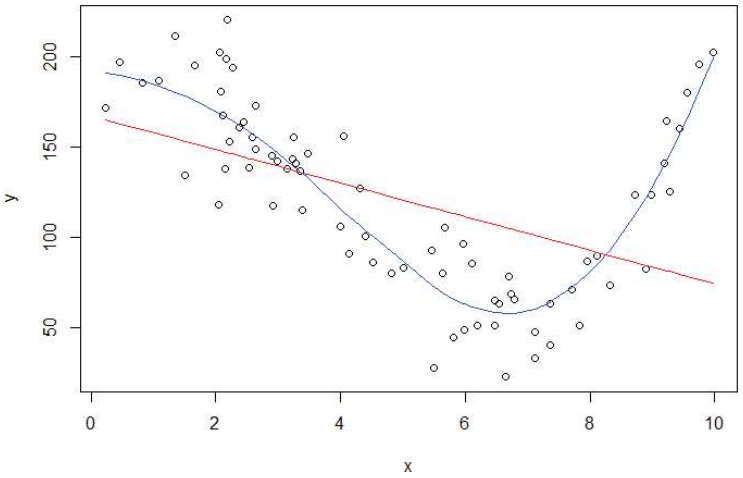

FIGURE 3-13 *Examining two variables with regression*

The R code to produce Figure 3-13 is as follows. The `runif(75,0,10)` generates 75 numbers between 0 to 10 with random deviates, and the numbers conform to the uniform distribution. The `rnorm(75,0,20)` generates 75 numbers that conform to the normal distribution, with the mean equal to 0 and the standard deviation equal to 20. The `points()` function is a generic function that draws a sequence of points at the specified coordinates. Parameter `type="l"` tells the function to draw a solid line. The `col` parameter sets the color of the line, where 2 represents the red color and 4 represents the blue color.

```
# 75 numbers between 0 and 10 of uniform distribution
x <- runif(75, 0, 10)

x <- sort(x)
y <- 200 + x^3 - 10 * x^2 + x + rnorm(75, 0, 20)


lr <- lm(y ~ x)        # linear regression
poly <- loess(y ~ x)   # LOESS
```

```
fit <- predict(poly)  # fit a nonlinear line

plot(x,y)

# draw the fitted line for the linear regression
points(x, lr$coefficients[1] + lr$coefficients[2] * x,
       type = "l", col = 2)

# draw the fitted line with LOESS
points(x, fit, type = "l", col = 4)
```

Dotchart and Barplot

Dotchart and barplot from the previous section can visualize multiple variables. Both of them use color as an additional dimension for visualizing the data.

For the same `mtcars` dataset, Figure 3-14 shows a dotchart that groups vehicle cylinders at the y-axis and uses colors to distinguish different cylinders. The vehicles are sorted according to their MPG values. The code to generate Figure 3-14 is shown next.

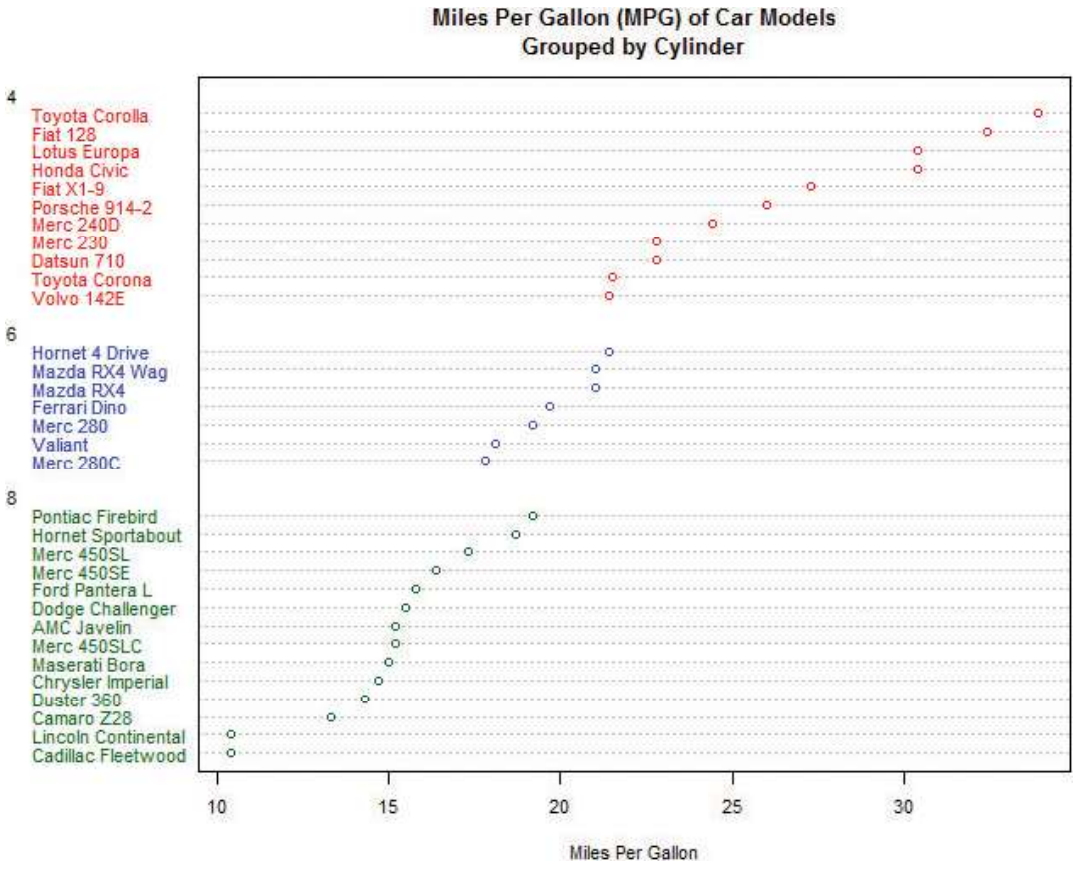

FIGURE 3-14 *Dotplot to visualize multiple variables*

```
# sort by mpg
cars <- mtcars[order(mtcars$mpg),]

# grouping variable must be a factor
cars$cyl <- factor(cars$cyl)

cars$color[cars$cyl==4] <- "red"
cars$color[cars$cyl==6] <- "blue"
cars$color[cars$cyl==8] <- "darkgreen"

dotchart(cars$mpg, labels=row.names(cars), cex=.7, groups= cars$cyl,
       main="Miles Per Gallon (MPG) of Car Models\nGrouped by Cylinder",
       xlab="Miles Per Gallon", color=cars$color, gcolor="black")
```

The barplot in Figure 3-15 visualizes the distribution of car cylinder counts and number of gears. The x-axis represents the number of cylinders, and the color represents the number of gears. The code to generate Figure 3-15 is shown next.

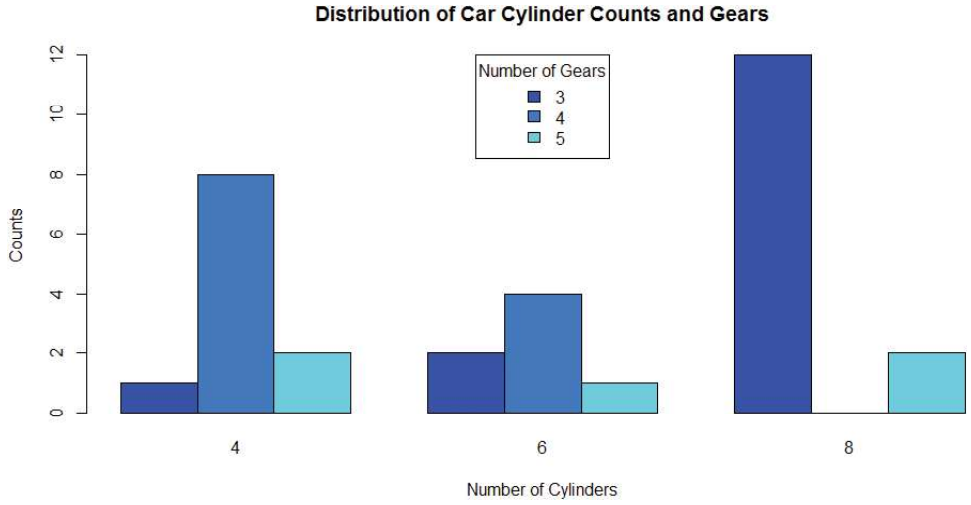

FIGURE 3-15 *Barplot to visualize multiple variables*

```
counts <- table(mtcars$gear, mtcars$cyl)
barplot(counts, main="Distribution of Car Cylinder Counts and Gears",
        xlab="Number of Cylinders", ylab="Counts",
        col=c("#0000FFFF", "#0080FFFF", "#00FFFFFF"),
        legend = rownames(counts), beside=TRUE,
        args.legend = list(x="top", title = "Number of Gears"))
```

Box-and-Whisker Plot

Box-and-whisker plots show the distribution of a continuous variable for each value of a discrete variable. The box-and-whisker plot in Figure 3-16 visualizes mean household incomes as a function of region in the United States. The first digit of the U.S. postal ("ZIP") code corresponds to a geographical region in the United States. In Figure 3-16, each data point corresponds to the mean household income from a particular zip code. The horizontal axis represents the first digit of a zip code, ranging from 0 to 9, where 0 corresponds to the northeast region of the United States (such as Maine, Vermont, and Massachusetts), and 9 corresponds to the southwest region (such as California and Hawaii). The vertical axis represents the logarithm of mean household incomes. The logarithm is taken to better visualize the distribution of the mean household incomes.

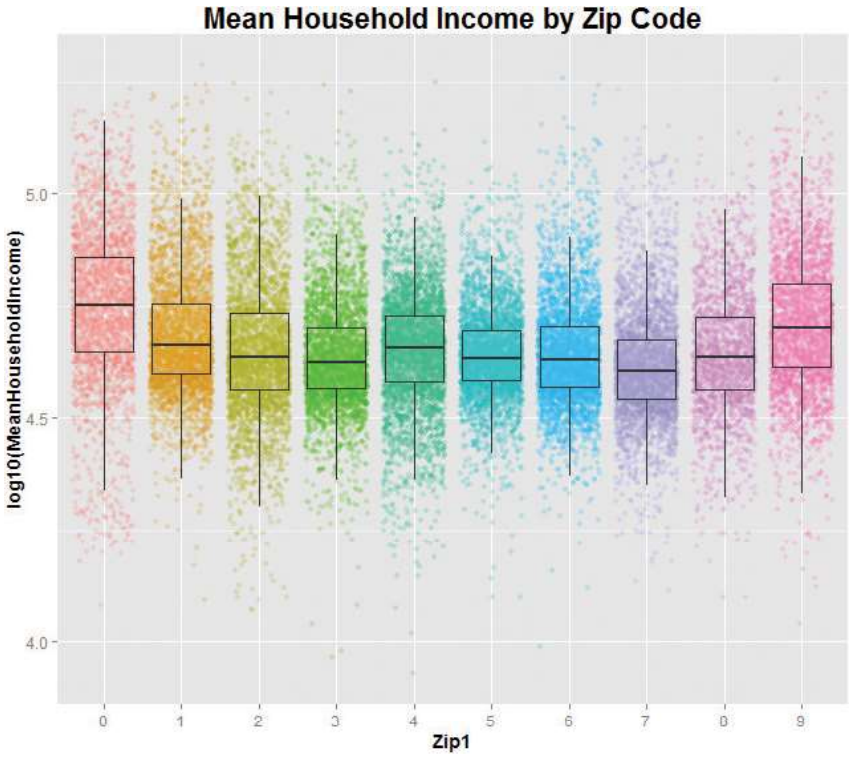

FIGURE 3-16 *A box-and-whisker plot of mean household income and geographical region*

In this figure, the scatterplot is displayed beneath the box-and-whisker plot, with some jittering for the overlap points so that each line of points widens into a strip. The "box" of the box-and-whisker shows the range that contains the central 50% of the data, and the line inside the box is the location of the median value. The upper and lower hinges of the boxes correspond to the first and third quartiles of the data. The upper whisker extends from the hinge to the highest value that is within 1.5 * IQR of the hinge. The lower whisker extends from the hinge to the lowest value within 1.5 * IQR of the hinge. IQR is the inter-quartile range, as discussed in Section 3.1.4. The points outside the whiskers can be considered possible outliers.

The graph shows how household income varies by region. The highest median incomes are in region 0 and region 9. Region 0 is slightly higher, but the boxes for the two regions overlap enough that the difference between the two regions probably is not significant. The lowest household incomes tend to be in region 7, which includes states such as Louisiana, Arkansas, and Oklahoma.

Assuming a data frame called `DF` contains two columns (`MeanHouseholdIncome` and `Zip1`), the following R script uses the `ggplot2` library [11] to plot a graph that is similar to Figure 3-16.

```
library(ggplot2)
# plot the jittered scatterplot w/ boxplot
# color-code points with zip codes
# the outlier.size=0 prevents the boxplot from plotting the outlier
ggplot(data=DF, aes(x=as.factor(Zip1), y=log10(MeanHouseholdIncome))) +
  geom_point(aes(color=factor(Zip1)), alpha=0.2, position="jitter") +
  geom_boxplot(outlier.size=0, alpha=0.1) +
  guides(colour=FALSE) +
  ggtitle ("Mean Household Income by Zip Code")
```

Alternatively, one can create a simple box-and-whisker plot with the `boxplot()` function provided by the R base package.

Hexbinplot for Large Datasets

This chapter has shown that scatterplot as a popular visualization can visualize data containing one or more variables. But one should be careful about using it on high-volume data. If there is too much data, the structure of the data may become difficult to see in a scatterplot. Consider a case to compare the logarithm of household income against the years of education, as shown in Figure 3-17. The cluster in the scatterplot on the left (a) suggests a somewhat linear relationship of the two variables. However, one cannot really see the structure of how the data is distributed inside the cluster. This is a Big Data type of problem. Millions or billions of data points would require different approaches for exploration, visualization, and analysis.

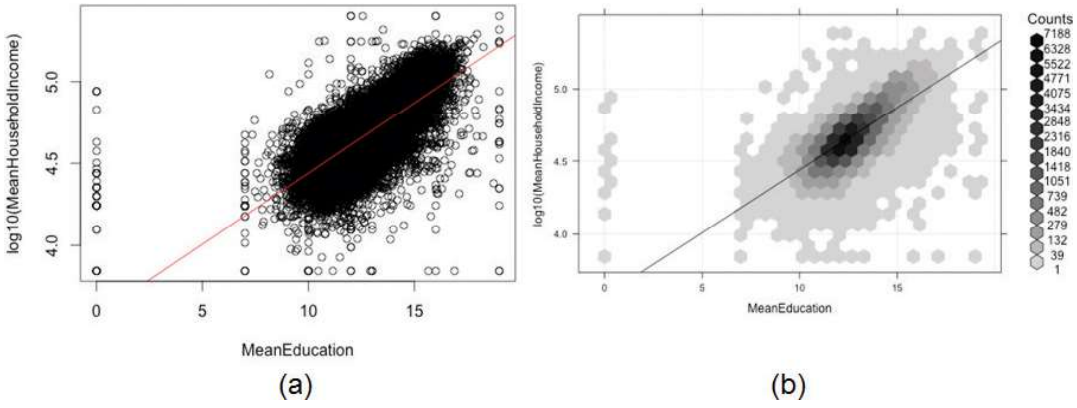

FIGURE 3-17 *(a) Scatterplot and (b) Hexbinplot of household income against years of education*

Although color and transparency can be used in a scatterplot to address this issue, a hexbinplot is sometimes a better alternative. A hexbinplot combines the ideas of scatterplot and histogram. Similar to a scatterplot, a hexbinplot visualizes data in the *x*-axis and *y*-axis. Data is placed into hexbins, and the third dimension uses shading to represent the concentration of data in each hexbin.

In Figure 3-17(b), the same data is plotted using a hexbinplot. The hexbinplot shows that the data is more densely clustered in a streak that runs through the center of the cluster, roughly along the regression line. The biggest concentration is around 12 years of education, extending to about 15 years.

In Figure 3-17, note the outlier data at `MeanEducation=0`. These data points may correspond to some missing data that needs further cleansing.

Assuming the two variables *`MeanHouseholdIncome`* and *`MeanEducation`* are from a data frame named *`zcta`*, the scatterplot of Figure 3-17(a) is plotted by the following R code.

```
# plot the data points
plot(log10(MeanHouseholdIncome) ~ MeanEducation, data=zcta)
# add a straight fitted line of the linear regression
abline(lm(log10(MeanHouseholdIncome) ~ MeanEducation, data=zcta), col='red')
```

Using the *`zcta`* data frame, the hexbinplot of Figure 3-17(b) is plotted by the following R code. Running the code requires the use of the `hexbin` package, which can be installed by running `install.packages("hexbin")`.

```
library(hexbin)
# "g" adds the grid, "r" adds the regression line
# sqrt transform on the count gives more dynamic range to the shading
# inv provides the inverse transformation function of trans
hexbinplot(log10(MeanHouseholdIncome) ~ MeanEducation,
  data=zcta, trans = sqrt, inv = function(x) x^2, type=c("g", "r"))
```

Scatterplot Matrix

A scatterplot matrix shows many scatterplots in a compact, side-by-side fashion. The scatterplot matrix, therefore, can visually represent multiple attributes of a dataset to explore their relationships, magnify differences, and disclose hidden patterns.

Fisher's *`iris`* dataset [13] includes the measurements in centimeters of the sepal length, sepal width, petal length, and petal width for 50 flowers from three species of iris. The three species are *setosa*, *versicolor*, and *virginica*. The iris dataset comes with the standard R distribution.

In Figure 3-18, all the variables of Fisher's iris dataset (sepal length, sepal width, petal length, and petal width) are compared in a scatterplot matrix. The three different colors represent three species of iris flowers. The scatterplot matrix in Figure 3-18 allows its viewers to compare the differences across the iris species for any pairs of attributes.

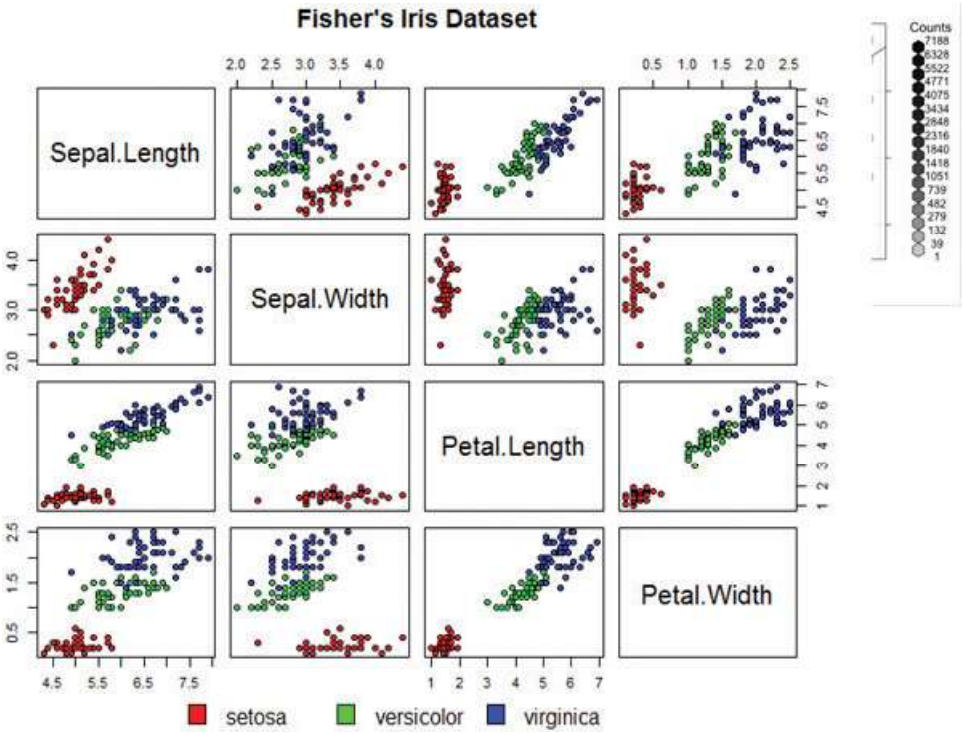

FIGURE 3-18 *Scatterplot matrix of Fisher's [13] iris dataset*

Consider the scatterplot from the first row and third column of Figure 3-18, where sepal length is compared against petal length. The horizontal axis is the petal length, and the vertical axis is the sepal length. The scatterplot shows that *versicolor* and *virginica* share similar sepal and petal lengths, although the latter has longer petals. The petal lengths of all *setosa* are about the same, and the petal lengths are remarkably shorter than the other two species. The scatterplot shows that for *versicolor* and *virginica*, sepal length grows linearly with the petal length.

The R code for generating the scatterplot matrix is provided next.

```
# define the colors
colors <- c("red", "green", "blue")

# draw the plot matrix
pairs(iris[1:4], main = "Fisher's Iris Dataset",
      pch = 21, bg = colors[unclass(iris$Species)] )

# set graphical parameter to clip plotting to the figure region
par(xpd = TRUE)

# add legend
legend(0.2, 0.02, horiz = TRUE, as.vector(unique(iris$Species)),
       fill = colors, bty = "n")
```

The vector `colors` defines the color scheme for the plot. It could be changed to something like `colors <- c("gray50", "white", "black")` to make the scatterplots grayscale.

Analyzing a Variable over Time

Visualizing a variable over time is the same as visualizing any pair of variables, but in this case the goal is to identify time-specific patterns.

Figure 3-19 plots the monthly total numbers of international airline passengers (in thousands) from January 1940 to December 1960. Enter `plot(AirPassengers)` in the R console to obtain a similar graph. The plot shows that, for each year, a large peak occurs mid-year around July and August, and a small peak happens around the end of the year, possibly due to the holidays. Such a phenomenon is referred to as a ***seasonality effect***.

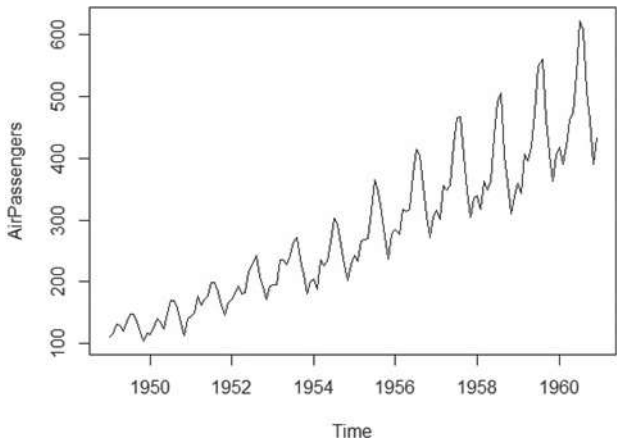

FIGURE 3-19 *Airline passenger counts from 1949 to 1960*

Additionally, the overall trend is that the number of air passengers steadily increased from 1949 to 1960. Chapter 8, "Advanced Analytical Theory and Methods: Time Series Analysis," discusses the analysis of such datasets in greater detail.

3.2.5 Data Exploration Versus Presentation

Using visualization for data exploration is different from presenting results to stakeholders. Not every type of plot is suitable for all audiences. Most of the plots presented earlier try to detail the data as clearly as possible for data scientists to identify structures and relationships. These graphs are more technical in nature and are better suited to technical audiences such as data scientists. Nontechnical stakeholders, however, generally prefer simple, clear graphics that focus on the message rather than the data.

Figure 3-20 shows the density plot on the distribution of account values from a bank. The data has been converted to the $\log_{10}$ scale. The plot includes a rug on the bottom to show the distribution of the variable. This graph is more suitable for data scientists and business analysts because it provides information that

can be relevant to the downstream analysis. The graph shows that the transformed account values follow an approximate normal distribution, in the range from $100 to $10,000,000. The median account value is approximately $30,000 ($10^{4.5}$), with the majority of the accounts between $1,000 ($10^3$) and $1,000,000 ($10^6$).

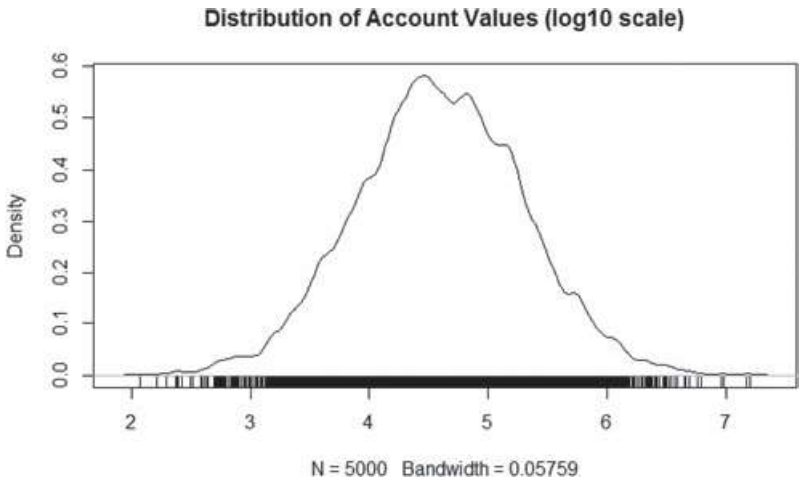

FIGURE 3-20 *Density plots are better to show to data scientists*

Density plots are fairly technical, and they contain so much information that they would be difficult to explain to less technical stakeholders. For example, it would be challenging to explain why the account values are in the $\log_{10}$ scale, and such information is not relevant to stakeholders. The same message can be conveyed by partitioning the data into log-like bins and presenting it as a histogram. As can be seen in Figure 3-21, the bulk of the accounts are in the $1,000–1,000,000 range, with the peak concentration in the $10–50K range, extending to $500K. This portrayal gives the stakeholders a better sense of the customer base than the density plot shown in Figure 3-20.

Note that the bin sizes should be carefully chosen to avoid distortion of the data. In this example, the bins in Figure 3-21 are chosen based on observations from the density plot in Figure 3-20. Without the density plot, the peak concentration might be just due to the somewhat arbitrary appearing choices for the bin sizes.

This simple example addresses the different needs of two groups of audience: analysts and stakeholders. Chapter 12, "The Endgame, or Putting It All Together," further discusses the best practices of delivering presentations to these two groups.

Following is the R code to generate the plots in Figure 3-20 and Figure 3-21.

```
# Generate random log normal income data
income = rlnorm(5000, meanlog=log(40000), sdlog=log(5))

# Part I: Create the density plot
plot(density(log10(income), adjust=0.5),
     main="Distribution of Account Values (log10 scale)")
# Add rug to the density plot
```

```
rug(log10(income))

# Part II: Make the histogram
# Create "log-like bins"
breaks = c(0, 1000, 5000, 10000, 50000, 100000, 5e5, 1e6, 2e7)
# Create bins and label the data
bins = cut(income, breaks, include.lowest=T,
           labels = c("< 1K", "1-5K", "5-10K", "10-50K",
                      "50-100K", "100-500K", "500K-1M", "> 1M"))
# Plot the bins
plot(bins, main = "Distribution of Account Values",
     xlab = "Account value ($ USD)",
     ylab = "Number of Accounts", col="blue")
```

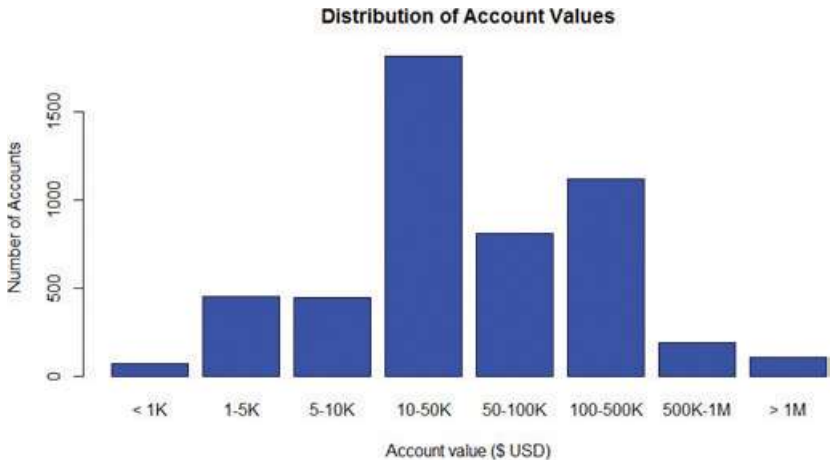

FIGURE 3-21 *Histograms are better to show to stakeholders*

3.3 Statistical Methods for Evaluation

Visualization is useful for data exploration and presentation, but statistics is crucial because it may exist throughout the entire Data Analytics Lifecycle. Statistical techniques are used during the initial data exploration and data preparation, model building, evaluation of the final models, and assessment of how the new models improve the situation when deployed in the field. In particular, statistics can help answer the following questions for data analytics:

- Model Building and Planning
  - What are the best input variables for the model?
  - Can the model predict the outcome given the input?

- Model Evaluation
  - Is the model accurate?
  - Does the model perform better than an obvious guess?
  - Does the model perform better than another candidate model?
- Model Deployment
  - Is the prediction sound?
  - Does the model have the desired effect (such as reducing the cost)?

This section discusses some useful statistical tools that may answer these questions.

3.3.1 Hypothesis Testing

When comparing populations, such as testing or evaluating the difference of the means from two samples of data (Figure 3-22), a common technique to assess the difference or the significance of the difference is ***hypothesis testing***.

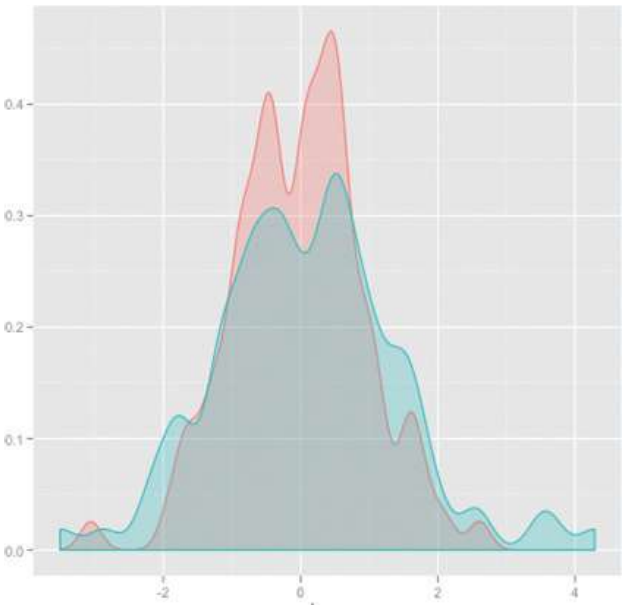

FIGURE 3-22 *Distributions of two samples of data*

The basic concept of hypothesis testing is to form an assertion and test it with data. When performing hypothesis tests, the common assumption is that there is no difference between two samples. This assumption is used as the default position for building the test or conducting a scientific experiment. Statisticians refer to this as the ***null hypothesis*** (H_0). The ***alternative hypothesis*** (H_A) is that there is a

difference between two samples. For example, if the task is to identify the effect of drug A compared to drug B on patients, the null hypothesis and alternative hypothesis would be this.

- H_0: Drug A and drug B have the same effect on patients.
- H_A: Drug A has a greater effect than drug B on patients.

If the task is to identify whether advertising Campaign C is effective on reducing customer churn, the null hypothesis and alternative hypothesis would be as follows.

- H_0: Campaign C does not reduce customer churn better than the current campaign method.
- H_A: Campaign C does reduce customer churn better than the current campaign.

It is important to state the null hypothesis and alternative hypothesis, because misstating them is likely to undermine the subsequent steps of the hypothesis testing process. A hypothesis test leads to either rejecting the null hypothesis in favor of the alternative or not rejecting the null hypothesis.

Table 3-5 includes some examples of null and alternative hypotheses that should be answered during the analytic lifecycle.

TABLE 3-5 *Example Null Hypotheses and Alternative Hypotheses*

Application	Null Hypothesis	Alternative Hypothesis
Accuracy Forecast	Model X *does not predict* better than the existing model.	Model X *predicts* better than the existing model.
Recommendation Engine	Algorithm Y *does not produce* better recommendations than the current algorithm being used.	Algorithm Y *produces* better recommendations than the current algorithm being used.
Regression Modeling	This variable *does not affect* the outcome because its coefficient is *zero*.	This variable *affects* outcome because its coefficient is not *zero*.

Once a model is built over the training data, it needs to be evaluated over the testing data to see if the proposed model predicts better than the existing model currently being used. The null hypothesis is that the proposed model does not predict better than the existing model. The alternative hypothesis is that the proposed model indeed predicts better than the existing model. In accuracy forecast, the null model could be that the sales of the next month are the same as the prior month. The hypothesis test needs to evaluate if the proposed model provides a better prediction. Take a recommendation engine as an example. The null hypothesis could be that the new algorithm does not produce better recommendations than the current algorithm being deployed. The alternative hypothesis is that the new algorithm produces better recommendations than the old algorithm.

When evaluating a model, sometimes it needs to be determined if a given input variable improves the model. In regression analysis (Chapter 6), for example, this is the same as asking if the regression coefficient for a variable is zero. The null hypothesis is that the coefficient is zero, which means the variable does not have an impact on the outcome. The alternative hypothesis is that the coefficient is nonzero, which means the variable does have an impact on the outcome.

A common hypothesis test is to compare the means of two populations. Two such hypothesis tests are discussed in Section 3.3.2.

3.3.2 Difference of Means

Hypothesis testing is a common approach to draw inferences on whether or not the two populations, denoted *pop*1 and *pop*2, are different from each other. This section provides two hypothesis tests to compare the means of the respective populations based on samples randomly drawn from each population. Specifically, the two hypothesis tests in this section consider the following null and alternative hypotheses.

- H_0: $\mu_1 = \mu_2$
- H_A: $\mu_1 \neq \mu_2$

The μ_1 and μ_2 denote the population means of *pop*1 and *pop*2, respectively.

The basic testing approach is to compare the observed sample means, $\bar{X}_1$ and $\bar{X}_2$, corresponding to each population. If the values of $\bar{X}_1$ and $\bar{X}_2$ are approximately equal to each other, the distributions of $\bar{X}_1$ and $\bar{X}_2$ overlap substantially (Figure 3-23), and the null hypothesis is supported. A large observed difference between the sample means indicates that the null hypothesis should be rejected. Formally, the difference in means can be tested using Student's *t*-test or the Welch's *t*-test.

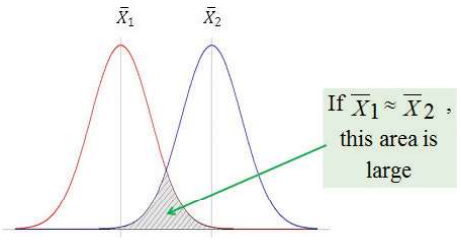

FIGURE 3-23 *Overlap of the two distributions is large if* $\bar{X}_1 \approx \bar{X}_2$

Student's t-test

Student's *t*-test assumes that distributions of the two populations have equal but unknown variances. Suppose n_1 and n_2 samples are randomly and independently selected from two populations, *pop*1 and *pop*2, respectively. If each population is normally distributed with the same mean ($\mu_1 = \mu_2$) and with the same variance, then *T* (the ***t-statistic***), given in Equation 3-1, follows a ***t-distribution*** with $n_1 + n_2 - 2$ ***degrees of freedom (df).***

$$T = \frac{\bar{X}_1 - \bar{X}_2}{S_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}}$$

where

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2} \qquad (3\text{-}1)$$

The shape of the *t*-distribution is similar to the normal distribution. In fact, as the degrees of freedom approaches 30 or more, the *t*-distribution is nearly identical to the normal distribution. Because the numerator of T is the difference of the sample means, if the observed value of T is far enough from zero such that the probability of observing such a value of T is unlikely, one would reject the null hypothesis that the population means are equal. Thus, for a small probability, say $\alpha = 0.05$, T^* is determined such that $P(|T| \geq T^*) = 0.05$. After the samples are collected and the observed value of T is calculated according to Equation 3-1, the null hypothesis ($\mu_1 = \mu_2$) is rejected if $|T| \geq T^*$.

In hypothesis testing, in general, the small probability, α, is known as the ***significance level*** of the test. The significance level of the test is the probability of rejecting the null hypothesis, when the null hypothesis is actually `TRUE`. In other words, for $\alpha = 0.05$, if the means from the two populations are truly equal, then in repeated random sampling, the observed magnitude of T would only exceed T^* 5% of the time.

In the following R code example, 10 observations are randomly selected from two normally distributed populations and assigned to the variables x and y. The two populations have a mean of 100 and 105, respectively, and a standard deviation equal to 5. Student's *t*-test is then conducted to determine if the obtained random samples support the rejection of the null hypothesis.

```
# generate random observations from the two populations
x <- rnorm(10, mean=100, sd=5)    # normal distribution centered at 100
y <- rnorm(20, mean=105, sd=5)    # normal distribution centered at 105

t.test(x, y, var.equal=TRUE)      # run the Student's t-test
Two Sample t-test

data:  x and y
t = -1.7828, df = 28, p-value = 0.08547
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  -6.1611557  0.4271893
sample estimates:
  mean of x mean of y
102.2136  105.0806
```

From the R output, the observed value of T is $t = -1.7828$. The negative sign is due to the fact that the sample mean of x is less than the sample mean of y. Using the `qt()` function in R, a T value of 2.0484 corresponds to a 0.05 significance level.

```
# obtain t value for a two-sided test at a 0.05 significance level
qt(p=0.05/2, df=28, lower.tail= FALSE)
2.048407
```

Because the magnitude of the observed T statistic is less than the T value corresponding to the 0.05 significance level ($|-1.7828| < 2.0484$), the null hypothesis is not rejected. Because the alternative hypothesis is that the means are not equal ($\mu_1 \neq \mu_2$), the possibilities of both $\mu_1 > \mu_2$ and $\mu_1 < \mu_2$ need to be considered. This form of Student's *t*-test is known as a ***two-sided hypothesis test***, and it is necessary for the sum of the probabilities under both tails of the *t*-distribution to equal the significance level. It is customary to evenly

divide the significance level between both tails. So, $p = 0.05/2 = 0.025$ was used in the `qt()` function to obtain the appropriate *t*-value.

To simplify the comparison of the *t*-test results to the significance level, the R output includes a quantity known as the ***p-value***. In the preceding example, the *p*-value is 0.08547, which is the sum of $P(T \leq -1.7828)$ and $P(T \geq 1.7828)$. Figure 3-24 illustrates the *t*-statistic for the area under the tail of a *t*-distribution. The *-t* and *t* are the observed values of the *t*-statistic. In the R output, $t = 1.7828$. The left shaded area corresponds to the $P(T \leq -1.7828)$, and the right shaded area corresponds to the $P(T \geq 1.7828)$.

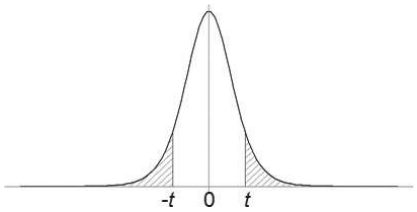

FIGURE 3-24 *Area under the tails (shaded) of a student's t-distribution*

In the R output, for a significance level of 0.05, the null hypothesis would not be rejected because the likelihood of a *T* value of magnitude 1.7828 or greater would occur at higher probability than 0.05. However, based on the *p*-value, if the significance level was chosen to be 0.10, instead of 0.05, the null hypothesis would be rejected. In general, the *p*-value offers the probability of observing such a sample result given the null hypothesis is `TRUE`.

A key assumption in using Student's *t*-test is that the population variances are equal. In the previous example, the `t.test()` function call includes `var.equal=TRUE` to specify that equality of the variances should be assumed. If that assumption is not appropriate, then Welch's *t*-test should be used.

Welch's t-test

When the equal population variance assumption is not justified in performing Student's *t*-test for the difference of means, Welch's *t*-test [14] can be used based on *T* expressed in Equation 3-2.

$$T_{welch} = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}} \tag{3-2}$$

where $\bar{X}_i$, S_i^2, and n_i correspond to the *i*-th sample mean, sample variance, and sample size. Notice that Welch's *t*-test uses the sample variance (S_i^2) for each population instead of the pooled sample variance.

In Welch's test, under the remaining assumptions of random samples from two normal populations with the same mean, the distribution of *T* is approximated by the *t*-distribution. The following R code performs the Welch's *t*-test on the same set of data analyzed in the earlier Student's *t*-test example.

```
t.test(x, y, var.equal=FALSE)       # run the Welch's t-test

Welch Two Sample t-test

data:  x and y
t = -1.6596, df = 15.118, p-value = 0.1176
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  -6.546629  0.812663
sample estimates:
  mean of x mean of y
102.2136  105.0806
```

In this particular example of using Welch's *t*-test, the *p*-value is 0.1176, which is greater than the *p*-value of 0.08547 observed in the Student's *t*-test example. In this case, the null hypothesis would not be rejected at a 0.10 or 0.05 significance level.

It should be noted that the degrees of freedom calculation is not as straightforward as in the Student's *t*-test. In fact, the degrees of freedom calculation often results in a non-integer value, as in this example. The degrees of freedom for Welch's *t*-test is defined in Equation 3-3.

$$df = \frac{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^2}{\frac{\left(\frac{S_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{S_2^2}{n_2}\right)^2}{n_2 - 1}} \tag{3-3}$$

In both the Student's and Welch's *t*-test examples, the R output provides 95% confidence intervals on the difference of the means. In both examples, the confidence intervals straddle zero. Regardless of the result of the hypothesis test, the confidence interval provides an interval estimate of the difference of the population means, not just a point estimate.

A ***confidence interval*** is an interval estimate of a population parameter or characteristic based on sample data. A confidence interval is used to indicate the uncertainty of a point estimate. If $\bar{x}$ is the estimate of some unknown population mean μ, the confidence interval provides an idea of how close $\bar{x}$ is to the unknown μ. For example, a 95% confidence interval for a population mean straddles the `TRUE`, but unknown mean 95% of the time. Consider Figure 3-25 as an example. Assume the confidence level is 95%. If the task is to estimate the mean of an unknown value μ in a normal distribution with known standard deviation σ and the estimate based on n observations is x, then the interval $\bar{x} \pm \frac{2\sigma}{\sqrt{n}}$ straddles the unknown value of μ with about a 95% chance. If one takes 100 different samples and computes the 95% confidence interval for the mean, 95 of the 100 confidence intervals will be expected to straddle the population mean μ.

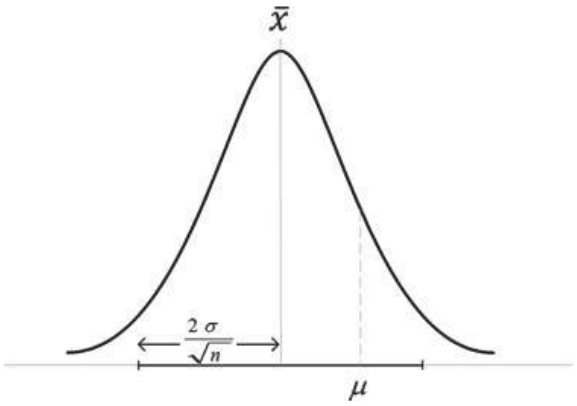

FIGURE 3-25 *A 95% confidence interval straddling the unknown population mean μ*

Confidence intervals appear again in Section 3.3.6 on ANOVA. Returning to the discussion of hypothesis testing, a key assumption in both the Student's and Welch's *t*-test is that the relevant population attribute is normally distributed. For non-normally distributed data, it is sometimes possible to transform the collected data to approximate a normal distribution. For example, taking the logarithm of a dataset can often transform skewed data to a dataset that is at least symmetric around its mean. However, if such transformations are ineffective, there are tests like the Wilcoxon rank-sum test that can be applied to see if two population distributions are different.

3.3.3 Wilcoxon Rank-Sum Test

A *t*-test represents a ***parametric test*** in that it makes assumptions about the population distributions from which the samples are drawn. If the populations cannot be assumed or transformed to follow a normal distribution, a ***nonparametric test*** can be used. The ***Wilcoxon rank-sum test*** [15] is a nonparametric hypothesis test that checks whether two populations are identically distributed. Assuming the two populations are identically distributed, one would expect that the ordering of any sampled observations would be evenly intermixed among themselves. For example, in ordering the observations, one would not expect to see a large number of observations from one population grouped together, especially at the beginning or the end of ordering.

Let the two populations again be `pop1` and `pop2`, with independently random samples of size n_1 and n_2 respectively. The total number of observations is then $N = n_1 + n_2$. The first step of the Wilcoxon test is to rank the set of observations from the two groups as if they came from one large group. The smallest observation receives a rank of 1, the second smallest observation receives a rank of 2, and so on with the largest observation being assigned the rank of *N*. Ties among the observations receive a rank equal to the average of the ranks they span. The test uses ranks instead of numerical outcomes to avoid specific assumptions about the shape of the distribution.

After ranking all the observations, the assigned ranks are summed for at least one population's sample. If the distribution of `pop1` is shifted to the right of the other distribution, the rank-sum corresponding to `pop1`'s sample should be larger than the rank-sum of `pop2`. The Wilcoxon rank-sum test determines the

significance of the observed rank-sums. The following R code performs the test on the same dataset used for the previous *t*-test.

```
wilcox.test(x, y, conf.int = TRUE)

Wilcoxon rank sum test

data:  x and y
W = 55, p-value = 0.04903
alternative hypothesis: true location shift is not equal to 0
95 percent confidence interval:
  -6.2596774 -0.1240618
sample estimates:
  difference in location
-3.417658
```

The `wilcox.test()` function ranks the observations, determines the respective rank-sums corresponding to each population's sample, and then determines the probability of such rank-sums of such magnitude being observed assuming that the population distributions are identical. In this example, the probability is given by the *p*-value of 0.04903. Thus, the null hypothesis would be rejected at a 0.05 significance level. The reader is cautioned against interpreting that one hypothesis test is clearly better than another test based solely on the examples given in this section.

Because the Wilcoxon test does not assume anything about the population distribution, it is generally considered more robust than the *t*-test. In other words, there are fewer assumptions to violate. However, when it is reasonable to assume that the data is normally distributed, Student's or Welch's *t*-test is an appropriate hypothesis test to consider.

3.3.4 Type I and Type II Errors

A hypothesis test may result in two types of errors, depending on whether the test accepts or rejects the null hypothesis. These two errors are known as type I and type II errors.

- A ***type I error*** is the rejection of the null hypothesis when the null hypothesis is `TRUE`. The probability of the type I error is denoted by the Greek letter α.
- A ***type II error*** is the acceptance of a null hypothesis when the null hypothesis is `FALSE`. The probability of the type II error is denoted by the Greek letter β.

Table 3-6 lists the four possible states of a hypothesis test, including the two types of errors.

TABLE 3-6 *Type I and Type II Error*

	H_0 is true	H_0 is false
H_0 is accepted	Correct outcome	*Type II Error*
H_0 is rejected	*Type I error*	Correct outcome

The significance level, as mentioned in the Student's *t*-test discussion, is equivalent to the type I error. For a significance level such as $\alpha = 0.05$, if the null hypothesis ($\mu_1 = \mu_2$) is TRUE, there is a 5% chance that the observed *T* value based on the sample data will be large enough to reject the null hypothesis. By selecting an appropriate significance level, the probability of committing a type I error can be defined before any data is collected or analyzed.

The probability of committing a Type II error is somewhat more difficult to determine. If two population means are truly not equal, the probability of committing a type II error will depend on how far apart the means truly are. To reduce the probability of a type II error to a reasonable level, it is often necessary to increase the sample size. This topic is addressed in the next section.

3.3.5 Power and Sample Size

The ***power*** of a test is the probability of correctly rejecting the null hypothesis. It is denoted by $1-\beta$, where β is the probability of a type II error. Because the power of a test improves as the sample size increases, power is used to determine the necessary sample size. In the difference of means, the power of a hypothesis test depends on the true difference of the population means. In other words, for a fixed significance level, a larger sample size is required to detect a smaller difference in the means. In general, the magnitude of the difference is known as the ***effect size***. As the sample size becomes larger, it is easier to detect a given effect size, δ, as illustrated in Figure 3-26.

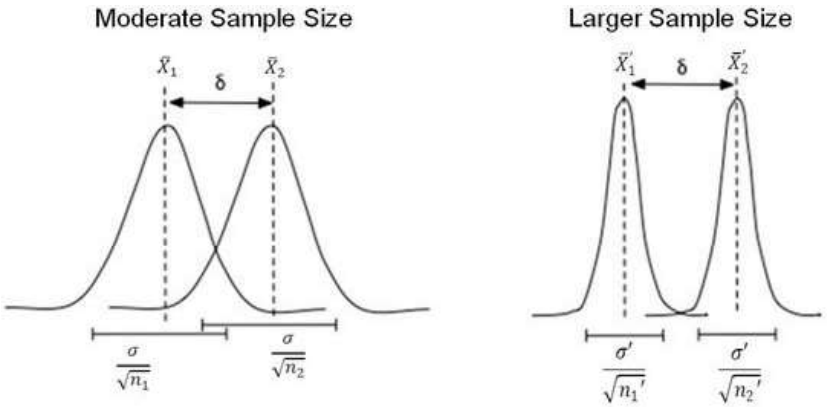

FIGURE 3-26 *A larger sample size better identifies a fixed effect size*

With a large enough sample size, almost any effect size can appear statistically significant. However, a very small effect size may be useless in a practical sense. It is important to consider an appropriate effect size for the problem at hand.

3.3.6 ANOVA

The *hypothesis tests* presented in the previous sections are good for analyzing means between two populations. But what if there are more than two populations? Consider an example of testing the impact of

nutrition and exercise on 60 candidates between age 18 and 50. The candidates are randomly split into six groups, each assigned with a different weight loss strategy, and the goal is to determine which strategy is the most effective.

- Group 1 only eats junk food.
- Group 2 only eats healthy food.
- Group 3 eats junk food and does cardio exercise every other day.
- Group 4 eats healthy food and does cardio exercise every other day.
- Group 5 eats junk food and does both cardio and strength training every other day.
- Group 6 eats healthy food and does both cardio and strength training every other day.

Multiple *t*-tests could be applied to each pair of weight loss strategies. In this example, the weight loss of Group 1 is compared with the weight loss of Group 2, 3, 4, 5, or 6. Similarly, the weight loss of Group 2 is compared with that of the next 4 groups. Therefore, a total of 15 *t*-tests would be performed.

However, multiple *t*-tests may not perform well on several populations for two reasons. First, because the number of *t*-tests increases as the number of groups increases, analysis using the multiple *t*-tests becomes cognitively more difficult. Second, by doing a greater number of analyses, the probability of committing at least one type I error somewhere in the analysis greatly increases.

Analysis of Variance (***ANOVA***) is designed to address these issues. ANOVA is a generalization of the hypothesis testing of the difference of two population means. ANOVA tests if any of the population means differ from the other population means. The null hypothesis of ANOVA is that all the population means are equal. The alternative hypothesis is that at least one pair of the population means is not equal. In other words,

- $\boldsymbol{H_0}$: $\mu_1 = \mu_2 = \ldots = \mu_n$
- $\boldsymbol{H_A}$: $\mu_i \neq \mu_j$ for at least one pair of i, j

As seen in Section 3.3.2, "Difference of Means," each population is assumed to be normally distributed with the same variance.

The first thing to calculate for the ANOVA is the test statistic. Essentially, the goal is to test whether the clusters formed by each population are more tightly grouped than the spread across all the populations.

Let the total number of populations be k. The total number of samples N is randomly split into the k groups. The number of samples in the i-th group is denoted as n_i, and the mean of the group is $\overline{X}_i$ where $i \in [1, k]$. The mean of all the samples is denoted as $\overline{X}_0$.

The ***between-groups mean sum of squares***, S_B^2, is an estimate of the ***between-groups variance***. It measures how the population means vary with respect to the grand mean, or the mean spread across all the populations. Formally, this is presented as shown in Equation 3-4.

$$S_B^2 = \frac{1}{k-1} \sum_{i=1}^{k} n_i \cdot (\overline{x}_i - \overline{x}_0)^2 \tag{3-4}$$

The ***within-group mean sum of squares***, S_W^2, is an estimate of the ***within-group variance***. It quantifies the spread of values within groups. Formally, this is presented as shown in Equation 3-5.

$$S_W^2 = \frac{1}{n-k}\sum_{i=1}^{k}\sum_{j=1}^{n_i}(x_{ij} - \bar{x}_i)^2 \tag{3-5}$$

If S_B^2 is much larger than S_W^2, then some of the population means are different from each other.

The *F*-test statistic is defined as the ratio of the between-groups mean sum of squares and the within-group mean sum of squares. Formally, this is presented as shown in Equation 3-6.

$$F = \frac{S_B^2}{S_W^2} \tag{3-6}$$

The *F*-test statistic in ANOVA can be thought of as a measure of how different the means are relative to the variability within each group. The larger the observed *F*-test statistic, the greater the likelihood that the differences between the means are due to something other than chance alone. The *F*-test statistic is used to test the hypothesis that the observed effects are not due to chance—that is, if the means are significantly different from one another.

Consider an example that every customer who visits a retail website gets one of two promotional offers or gets no promotion at all. The goal is to see if making the promotional offers makes a difference. ANOVA could be used, and the null hypothesis is that neither promotion makes a difference. The code that follows randomly generates a total of 500 observations of purchase sizes on three different offer options.

```
offers <- sample(c("offer1", "offer2", "nopromo"), size=500, replace=T)

# Simulated 500 observations of purchase sizes on the 3 offer options
purchasesize <- ifelse(offers=="offer1", rnorm(500, mean=80, sd=30),
                ifelse(offers=="offer2", rnorm(500, mean=85, sd=30),
                      rnorm(500, mean=40, sd=30)))

# create a data frame of offer option and purchase size
offertest <- data.frame(offer=as.factor(offers),
                        purchase_amt=purchasesize)
```

The summary of the *offertest* data frame shows that 170 `offer1`, 161 `offer2`, and 169 `nopromo` (no promotion) offers have been made. It also shows the range of purchase size (`purchase_amt`) for each of the three offer options.

```
# display a summary of offertest where offer="offer1"
summary(offertest[offertest$offer=="offer1",])
     offer      purchase_amt
 nopromo:  0   Min.   :  4.521
 offer1 :170   1st Qu.: 58.158
 offer2 :  0   Median : 76.944
               Mean   : 81.936
               3rd Qu.:104.959
               Max.   :180.507

# display a summary of offertest where offer="offer2"
summary(offertest[offertest$offer=="offer2",])
```

```
     offer      purchase_amt
 nopromo:  0   Min.   : 14.04
 offer1 :  0   1st Qu.: 69.46
 offer2 :161   Median : 90.20
               Mean   : 89.09
               3rd Qu.:107.48
               Max.   :154.33

# display a summary of offertest where offer="nopromo"
summary(offertest[offertest$offer=="nopromo",])
     offer      purchase_amt
 nopromo:169   Min.   :-27.00
 offer1 :  0   1st Qu.: 20.22
 offer2 :  0   Median : 42.44
               Mean   : 40.97
               3rd Qu.: 58.96
               Max.   :164.04
```

The `aov()` function performs the ANOVA on purchase size and offer options.

```
# fit ANOVA test
model <- aov(purchase_amt ~ offers, data=offertest)
```

The `summary()` function shows a summary of the model. The degrees of freedom for offers is 2, which corresponds to the $k-1$ in the denominator of Equation 3-4. The degrees of freedom for residuals is 497, which corresponds to the $n-k$ in the denominator of Equation 3-5.

```
summary(model)
             Df Sum Sq Mean Sq F value Pr(>F)
offers        2 225222  112611   130.6 <2e-16 ***
Residuals   497 428470     862
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The output also includes the S_B^2 (112,611), S_W^2 (862), the *F*-test statistic (130.6), and the *p*-value (< 2e–16). The *F*-test statistic is much greater than 1 with a *p*-value much less than 1. Thus, the null hypothesis that the means are equal should be rejected.

However, the result does not show whether `offer1` is different from `offer2`, which requires additional tests. The `TukeyHSD()` function implements Tukey's Honest Significant Difference (HSD) on all pair-wise tests for difference of means.

```
TukeyHSD(model)
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = purchase_amt ~ offers, data = offertest)

$offers
                    diff        lwr      upr     p adj
offer1-nopromo 40.961437 33.4638483 48.45903 0.0000000
```

```
offer2-nopromo 48.120286 40.5189446 55.72163 0.0000000
offer2-offer1   7.158849 -0.4315769 14.74928 0.0692895
```

The result includes *p*-values of pair-wise comparisons of the three offer options. The *p*-values for `offer1-nopromo` and `offer-nopromo` are equal to 0, smaller than the significance level 0.05. This suggests that both `offer1` and `offer2` are significantly different from `nopromo`. A *p*-value of 0.0692895 for `offer2` against `offer1` is greater than the significance level 0.05. This suggests that `offer2` is *not* significantly different from `offer1`.

Because only the influence of one factor (offers) was executed, the presented ANOVA is known as one-way ANOVA. If the goal is to analyze two factors, such as offers and day of week, that would be a two-way ANOVA [16]. If the goal is to model more than one outcome variable, then multivariate ANOVA (or MANOVA) could be used.

Summary

R is a popular package and programming language for data exploration, analytics, and visualization. As an introduction to R, this chapter covers the R GUI, data I/O, attribute and data types, and descriptive statistics. This chapter also discusses how to use R to perform exploratory data analysis, including the discovery of dirty data, visualization of one or more variables, and customization of visualization for different audiences. Finally, the chapter introduces some basic statistical methods. The first statistical method presented in the chapter is the hypothesis testing. The Student's *t*-test and Welch's *t*-test are included as two example hypothesis tests designed for testing the difference of means. Other statistical methods and tools presented in this chapter include confidence intervals, Wilcoxon rank-sum test, type I and II errors, effect size, and ANOVA.

Exercises

1. How many levels does *fdata* contain in the following R code?

```
data = c(1,2,2,3,1,2,3,3,1,2,3,3,1)
fdata = factor(data)
```

2. Two vectors, *v1* and *v2*, are created with the following R code:

```
v1 <- 1:5
v2 <- 6:2
```

   What are the results of `cbind(v1,v2)` and `rbind(v1,v2)`?

3. What R command(s) would you use to remove null values from a dataset?

4. What R command can be used to install an additional R package?

5. What R function is used to encode a vector as a category?

6. What is a rug plot used for in a density plot?

7. An online retailer wants to study the purchase behaviors of its customers. Figure 3-27 shows the density plot of the purchase sizes (in dollars). What would be your recommendation to enhance the plot to detect more structures that otherwise might be missed?