MSIS 5223: Tutorial 1 – Using Data in R

If you have Crawley's book *The R Book* you should read chapters 3 and 4 on data input and dataframes. Also, review the tutorial documents on D2L for R; specifically, *R – An Introduction to R* and *R – Data Import-Export*.

Instructions

The purpose of this tutorial is to help you become familiar with using R. The most fundamental functions in R involve importing data into R, manipulating that data, and understanding how to access that data. Once you become familiar with manipulating the data, you should have no problem familiarizing yourself with it.

If you wish to see examples of the code used in this tutorial, please follow along and read the R script *Data Derivation, Selection, ODBC*.

1. Setting the Working Directory

The working directory is the default directory R will save files to. All programs, including video games, contain a default directory. When you create a new Word or Excel file, the directory you saved it in becomes the default for that file. Have you ever noticed that Word creates temporary files while working on a Word document? This file is created in the default working directory.

While you do not have to change the working directory in R, I recommend doing so as it keeps your data organized. If you work on all of your projects within the same working directory, you will have a very full folder with many files not related to each other. What a mess!

It is very simple to set the working directory within R. The following shows you how to do this:

```
setwd("c:\\Users\\Username\\Documents\\MSIS 5633\\Semester Project\\Data")
```

You'll notice that you need to specify the full path of the directory for your file, starting from the root, which is usually the C-drive. You will also notice that instead of using a single backslash to delineate directories, R requires a double-backslash. For example, if you have a file contained in your Documents folder within Windows, you will need to add double-backslashes into the path like so:

```
c:\users\Person001\Documents\MSIS5633\Project\Data\expedia_reviews.txt
```

$$\downarrow$$

```
c:\\users\\Person001\\Documents\\MSIS5633\\Project\\Data\\expedia_reviews.txt
```

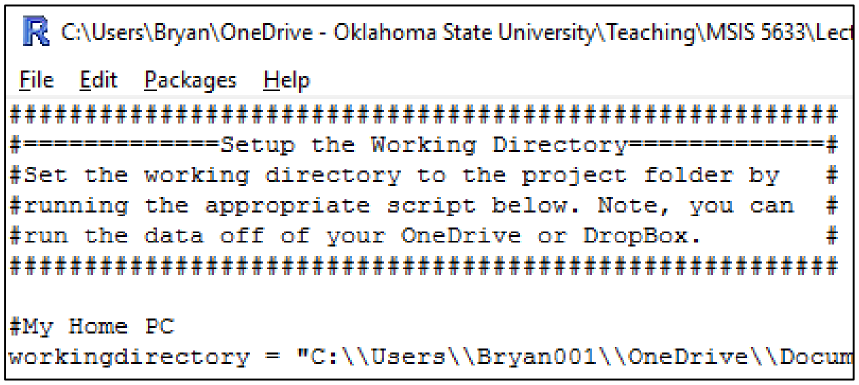
Be sure to put quotation marks around the path. Failure to do so will result in an error. Another method of setting your working directory is to create a character object that is the directory path. Then, simply reference that object in the function setwd().

```
workingdirectory = "c:\\Users\\jim\\OneDrive\\MSIS 5633\\Data"
setwd(workingdirectory)
```

The advantage of this is you can reference the working directory in R for functions beside setwd(). An example is provided in the next section of this Exercise.

One of the advantages of using R is it creates a history of your analytics project. If you forgo working on your project for months and return to it later, you only need to parse through your R code to remember what you did. Your working directory provides the location of where you stored your data files.

To further help clarify code for future comprehension, use comments as much as possible. Comments are limited to a single line and are designated with #. For example, here is code that I often use before I setup the working directory:


C:\Users\Bryan\OneDrive - Oklahoma State University\Teaching\MSIS 5633\Lect
File Edit Packages Help

```
#######################################################
#=============Setup the Working Directory=============#
#Set the working directory to the project folder by   #
#running the appropriate script below. Note, you can  #
#run the data off of your OneDrive or DropBox.        #
#######################################################

#My Home PC
workingdirectory = "C:\\Users\\Bryan001\\OneDrive\\Docum
```

Figure 1 Commenting within R

2. Data Input in R

By default, R contains a vast list of datasets designed to help you learn its various functions. To access this list, simply type data() in the console. You'll notice this list spans many pages. This is overwhelming, to say the least. If you would like to use a dataset specifically designed for a particular analysis, use the function try(). One of the libraries used this semester is *psych*, which contains many functions used for descriptive statistics. Try the following command:

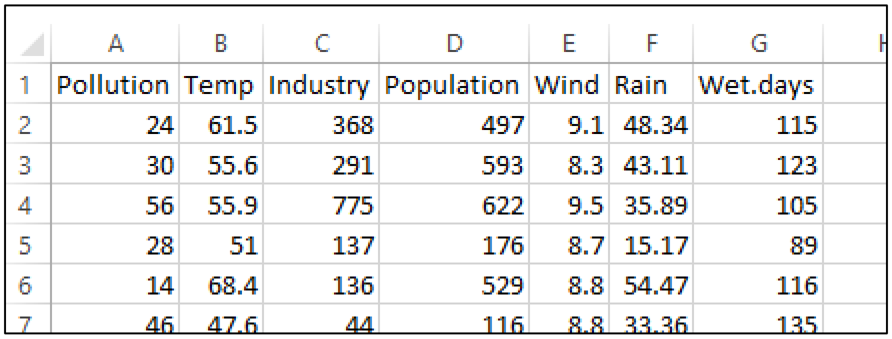
```
try(data(package="psych"))
```

You should see a short list of datasets such as Bechtoldt, Chen (Schmid), Holzinger, and others. Most libraries contain built-in data sets to help you learn the functions of the library. How thoughtful are the authors of those libraries?

Usually, you will not be using a built-in dataset when working with R. You have obtained your own and would like to perform an analysis on it. R offers several methods for opening data files. The most important function due to its robustness is read.table(). Often, the function will look like this:

datafile = read.table("c:\\directory-of-file\\filename.txt", header=T, sep="\t")

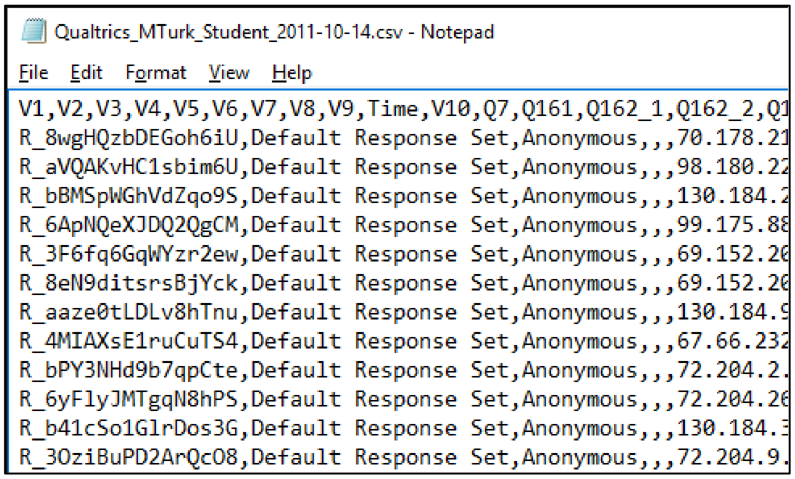
After specifying the location of the file you need to indicate whether the file contains a header row as well as the type of file you are working with. What is the header? Typically, it is the first row of a data file that provides the column names, as seen below in Excel. The column headings start with *Pollution* and end with *Wet.days*. Failure to include this will cause R to treat the column header row as just another row of data.



	A	B	C	D	E	F	G
1	Pollution	Temp	Industry	Population	Wind	Rain	Wet.days
2	24	61.5	368	497	9.1	48.34	115
3	30	55.6	291	593	8.3	43.11	123
4	56	55.9	775	622	9.5	35.89	105
5	28	51	137	176	8.7	15.17	89
6	14	68.4	136	529	8.8	54.47	116
7	46	47.6	44	116	8.8	33.36	135

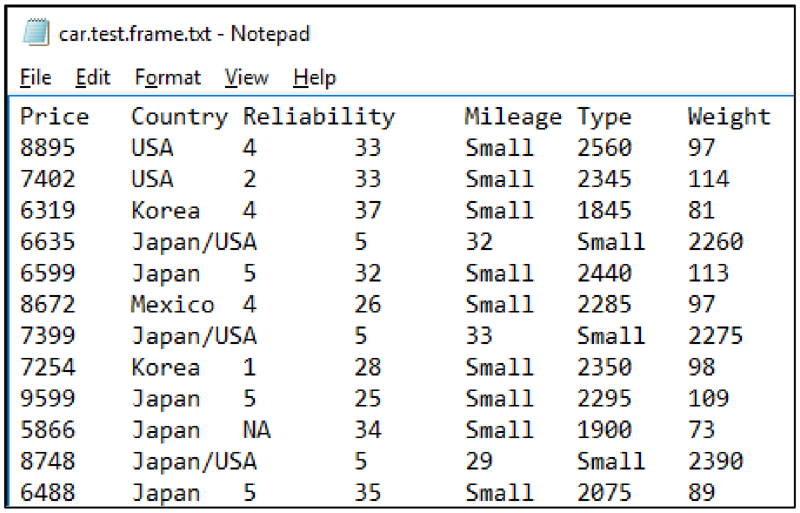
Figure 2 Column Headers

The last argument in the function deals with the how the file's columns are separated. The two most common types you will encounter are tab-delimited (files often end in txt) and comma-separated (files often end with csv). Below are examples of both types of files opened within Notepad.



```
V1,V2,V3,V4,V5,V6,V7,V8,V9,Time,V10,Q7,Q161,Q162_1,Q162_2,Q1
R_8wgHQzbDEGoh6iU,Default Response Set,Anonymous,,,70.178.21
R_aVQAKvHC1sbim6U,Default Response Set,Anonymous,,,98.180.22
R_bBMSpWGhVdZqo9S,Default Response Set,Anonymous,,,130.184.2
R_6ApNQeXJDQ2QgCM,Default Response Set,Anonymous,,,99.175.88
R_3F6fq6GqWYzr2ew,Default Response Set,Anonymous,,,69.152.20
R_8eN9ditsrsBjYck,Default Response Set,Anonymous,,,69.152.20
R_aaze0tLDLv8hTnu,Default Response Set,Anonymous,,,130.184.9
R_4MIAXsE1ruCuTS4,Default Response Set,Anonymous,,,67.66.232
R_bPY3NHd9b7qpCte,Default Response Set,Anonymous,,,72.204.2.
R_6yFlyJMTgqN8hPS,Default Response Set,Anonymous,,,72.204.26
R_b41cSo1GlrDos3G,Default Response Set,Anonymous,,,130.184.3
R_3OziBuPD2ArQc08,Default Response Set,Anonymous,,,72.204.9.
```

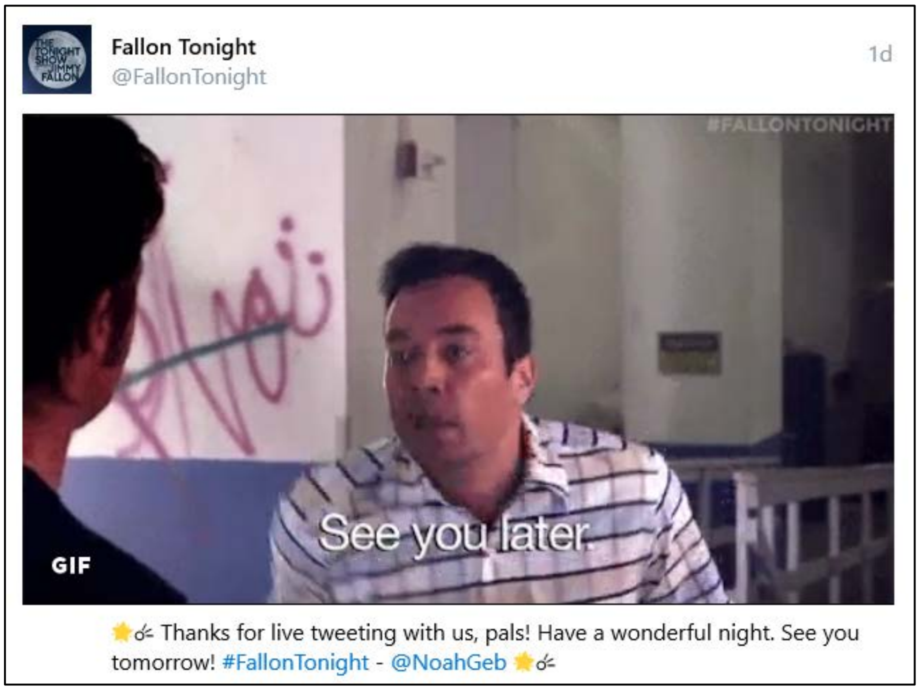
Figure 3 Comma Separated File



```
Price   Country Reliability     Mileage Type    Weight
8895    USA     4       33      Small   2560    97
7402    USA     2       33      Small   2345    114
6319    Korea   4       37      Small   1845    81
6635    Japan/USA       5       32      Small   2260
6599    Japan   5       32      Small   2440    113
8672    Mexico  4       26      Small   2285    97
7399    Japan/USA       5       33      Small   2275
7254    Korea   1       28      Small   2350    98
9599    Japan   5       25      Small   2295    109
5866    Japan   NA      34      Small   1900    73
8748    Japan/USA       5       29      Small   2390
6488    Japan   5       35      Small   2075    89
```

Figure 4 Tab-Delimited File

Right away you will notice differences in how these two types of files are formatted. A comma separated file contains lots of commas. This can be problematic if you have a column of data that is text. For example, if you are pulling data from Twitter you will have tweets that contain commas (see example below). If the file was not set up correctly, that comma in the text will be treated as a separator and you will end up with one extra column for that row.


Fallon Tonight
@FallonTonight
1d

⭐ Thanks for live tweeting with us, pals! Have a wonderful night. See you tomorrow! #FallonTonight - @NoahGeb ⭐

Figure 5 Tweet with Commas

How do you differentiate the file type using `read.table()`? If you are using a tab-delimited file you need to use `\t` to represent tabs, and a comma for comma separated files; they would look like `sep="\t"` and `sep=","` respectively.

When importing data you may encounter errors or problems with importing your file. Some common reasons may be due to your column headers having incorrect formatting such as blank spaces. For example, if you have a column header called *annual sales* you should convert it to *annualsales*, *annual_sales*, or *annual.sales*. If your headers have blank spaces, then R will assume these are two separate columns. So, *annual sales* is treated as two columns, not one. This results in your header having one more column than the body of data!

Another common error encountered in R is due to missing values in your dataset. This is not to say R cannot handle missing values, but those missing values cannot be left as a null value. The easiest solution to this is to open your file in Excel and replace all missing values with the value NA. To R, NA represents a null value. In fact, all database systems use a value to represent a missing value. While this seems odd at first glance, it is logical. A computer must represent the lack of data somehow in its memory; it cannot merely have nothing.

R has two alternate functions for opening files that you should be aware of. The first one allows you to browse for the file of interest. It is possible that you may not remember the exact location of your file and wish to browse your computer within R. To do so, simply type the following command:

```
datafile2 = read.table(file.choose(), header=T)
```

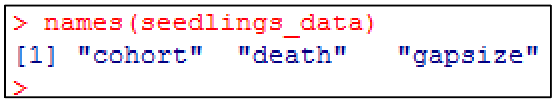
The other function that you should be aware of is scan(). This is much more flexible to use than read.table(). This also means it is much more difficult to use. Why would you use this function? One good reason is your data file contains different numbers of values for each record. Scanning the file will allow you to import that kind of file while read.table() demands each row have equal columns.

Let's return to the function read.table(). The last item to mention is saving the data file as a dataframe object within R. A dataframe is a technical, fancy way of saying a dataset in R. By saving it as an object you can refer to it and its many parts anytime you would like to. Here is the code one more time:

```
datafile = read.table("c:\\directory-of-file\\filename.txt", header=T, sep="\t")
```

The dataframe object we have chosen to create is called *datafile*. You could have named this *banana*, *weak_pancake*, *weakpancakge*, or *fifteen.toes_green*. I like to use periods or underscores to separate words. You cannot use spaces. Keep in mind, if you already have a dataframe or variable with a name, don't create a new dataframe with that exact same name! This will overwrite the previous value with the new value, which is a dataframe in this case.

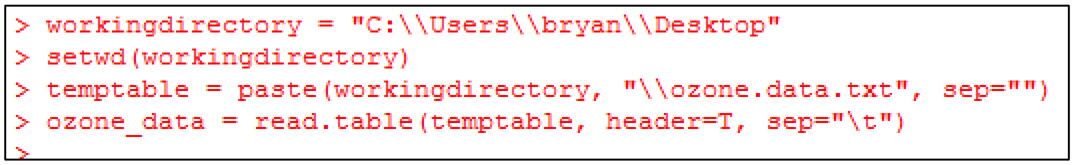
The last thing you should know is how to reference or access the columns within your dataframe. Simply use a dollar sign attached to the dataframe object to do so. For example, if I have the column header *annual_income* in my dataframe *sales_data*, I would simply type sales_data$annual_income. If you are not sure what the column header names are, use the names() function to find out.



```
> names(seedlings_data)
[1] "cohort"   "death"    "gapsize"
>
```

Figure 6 Using the names() Function

In the previous section I suggested you could save your working directory as a character object to reference later in R. One way I do that is with read.table(). Instead of typing out the entire path of the file's location, I can simply reference the variable I created containing the path. See below for an example.



```
> workingdirectory = "C:\\Users\\bryan\\Desktop"
> setwd(workingdirectory)
> temptable = paste(workingdirectory, "\\ozone.data.txt", sep="")
> ozone_data = read.table(temptable, header=T, sep="\t")
>
```

Figure 7 Referencing the Working Directory

3. Changing Column Data

Removing columns in R is a straightforward process. After importing your data into a dataframe, use the function subset() to remove the columns of interest. If you are using an automobile dataset with a list of cars with their respective characteristics, you may wish to remove columns that you are not interested in. The dataset contains the following variables, in order: *weight*, *total_mileage*, *transmission*, *gps_bt*, *style*, *color*, *engine*, and *manufacturer*.

In this situation, you are not interested in the column *total_mileage*, *transmission*, and *gps_bt*. You own a fifth-wheel and use your phone for navigation. To remove the columns, type the following:

```
car_data = subset(car_data, select=-c(total_mileage, transmission, gps_bt))
```

or

```
car_data = subset(car_data, select=-c(total_mileage:gps_bt))
```

The first example is a way to remove individual columns of data. If you have a long list of columns to remove, this list would become long. The second example shows a way in which to remove consecutive columns, giving the first, *total_mileage*, and the last, *gps_bt*. Note, the second method will remove any columns found between the two specified columns.

Renaming columns is a simple process. R provides many possible ways to change a column header. The first requires you to type in all of the column names, even the ones you are not changing. This can be tedious, however, if you have a lot of columns. If that is the case, the second method is better; also, it is recommended that you remove unwanted columns or create a subset of your data prior to renaming columns. Using the previous example, you want to rename the columns *gps_bt* and *engine* in your car data. You would type the following:

```
colnames(car_data) = c("total_mileage", "transmission", "gpsnav_bluetooth", "style", "color", "cylinder_type", "manufacturer")
```

The second method relies on knowing the index value of your column. That is, which position is your column in? One simple way to find out is type in names(*dataframe*) where *dataframe* is your dataframe object. Count each column header listed until you find yours. If your column is the fourth, you would type in the following, where *dataframe* is your dataframe object, 4 is the index value of your column, and NewName is the text you want to name your column:

```
names(dataframe)[4] = “NewName”
```

In addition to changing the columns within your dataframe, you may wish to change the actual data within your columns. For example, say you have a column that contains race information called *race_primary*. You find that your data has two versions of a null value: Null and NULL. The majority of stat programs will recognize these as two completely different values. You decide you would like to change all the uppercase values of “NULL” to match that of “Null.” To do this, simply use the code below:

```
gsub("NULL", "Null", dataframe$race_primary)
```

The function gsub() is used for pattern matching and replacement. The first argument in the above example—“NULL”—is the value you want to change. The second argument—“Null”—is the new value you would like. The last argument is the object you wish to change, which in this case is a column within a dataframe. Obviously, you would use the name of your dataframe and the name of your column instead of those in the example.

To further illustrate how this function works, assume you would like to change all values of “Null” and “NULL” to NA. You would have to run the function twice as shown below:

```
gsub("NULL", "NA", dataframe$race_primary)
gsub("Null", "NA", dataframe$race_primary)
```

Sometimes, you have more complex changes for your data in mind. For example, you may need to convert the first letter of all the values in your variable to a capital O. Or, perhaps you need to change the third letter for all values that begin with a capital T, but end with a lowercase W, but not if it contains the letter F as the third or fourth letter. This requires the use of a technique referred to as *regular expressions*.

This is beyond the scope of this class because of its level of technicality and sophistication. If you are interested, however, you can read Chapter 2 of Crawley's *The R Book* (2nd Edition), page 97.

4. Working with a Dataframe

This next section deals with working with the data within a dataframe. Prior to even performing simple descriptive statistics, it may be beneficial to familiarize yourself with the data contained in the dataframe. This entails selecting specific columns or rows, sorting data, and selecting data based on conditions.

Sometimes it is useful to select specific values within your data. R uses an indexing system, like the majority of statistical packages, for both rows and columns. In certain statistical packages with a GUI (graphical user interface), like IBM Modeler or SAS Enterprise Guide, you navigate a spreadsheet-like interface and use your mouse to highlight a specific value or values in the dataset. In R, because there is no GUI, you must specify the value using the indices, or subscripts as they are sometimes called.

The indices look like this

[*r*, *c*]

where *r* is the row value and *c* is the column value. Recall the car data contains eight columns of data. If you want to find the data for the fourth row of data, for transmission (which is the third column), you would type the following:

```
car_data[4,3]
```

What if you cannot remember which order the columns are in? You can use the `name()` function to list the columns, in order, in your dataset. It is also possible to pull the data for a single row. To do so, you would leave the column index value blank, like so:

```
car_data[36,]
```

This brings up all of the data for row 36 for all columns of data. Likewise, you can pull data for a single column for all rows of data. If you would like to pull data for *style*, then you would leave the row index value blank, like so:

```
car_data[,5]
```

It is quite possible that you would like specific columns for a specific row. Assume you would like only columns 2, 3, and 4 for row 107. There are two ways to do this. The first method uses a range of columns, all inclusive; the second method specifies each column using indices or column names:

Method 1: `car_data[107,2:4]`

Method 2a: `car_data[107,c(2,3,4)]`

Method 2b: car_data[107,c("*total_mileage*", "*transmission*", "*gps_bt*")]

The first method is useful if you have a long range of columns and you do not want to type out each one individually. The second method gives you the freedom to specify the columns of interest. If you only wanted columns 2 and 4 without 3, you could not use the first method; only the second method would allow this:

```
car_data[107,c(2,4)]
car_data[107,c("total_mileage", "gps_bt")]
```

You can also save an entire column of data as a new dataframe like so:

```
gps_data = car_data[,4]
gps_data = car_data[,c("gps_bt")]
```

Of course, a simple way of doing this would be referencing the column of data from the dataframe itself. In R, you can reference any column of data within a dataframe simply by using the $ symbol to designate a column. For example, to obtain the data for *total_mileage* you would type the following into R:

```
car_data$total_mileage
```

This returns only the data for the column *total_mileage*. This is useful for referencing the column to be used in various equations and functions. If you wanted to calculate the mean value of *total_mileage*, you would type the following:

```
mean(car_data$total_mileage)
```

Likewise, if you wanted to create a new dataframe that only contains a single column of data, you would type in the following command:

```
gps_data = car_data$gps_bt
```

This will only work for a single column of data. If you would like to save two or more columns of data into a new dataframe, you would have to use the previously shown methods of referencing multiple columns.

Three more functions that are useful should be noted here. The first one identifies the unique values contained within your data. If you have categorical data, this can be helpful in determining the various values contained in your variable (in R, the term "factor" is used instead of categorical). For example, assume that *color* contains five unique values: red, blue, green, white, and black. To see a list of these values, provide R with the following code:

```
unique(car_data$color)
```

The second function, or set of functions, provides basic information on row and column size. This may be useful if you do not know the number of records your data contains or the number of columns. Respectively, these are provided as follows:

```
nrow(dataframe)
ncol(dataframe)
```

The third set of functions allows you to sort your data. You can sort ascending, descending, select multiple columns to sort by, include only certain columns in your results, and many other combinations. A simple sort would look like this:

```
car_data[order(car_data$gps_bt),]
```

This sorts the data only on the column *gps_bt*. If you would like to reverse the sort, use the following code:

```
car_data[rev(order(car_data$gps_bt)),]
```

Sorting multiple columns is also straight forward. Just add the additional columns, in the order that you would like them sorted. If you want to sort by *gps_bt* and *transmission* in that order, you would list *gps_bt* first; if you want *transmission* sorted first, then list *transmission* first. See the following for the examples:

```
car_data[order(car_data$gps_bt, car_data$transmission),]
car_data[order(car_data$transmission, car_data$gps_bt),]
```

Observe that the sorting code is entered in the index for row; no column index was specified. One of the issues in sorting the data this way is R includes all of the other columns in your dataset. What if you don't want all the other columns included in your sort?

You can include the column index value of the column you want to retain. Here is an example of doing this, including an alternative method for choosing the columns in your dataset:

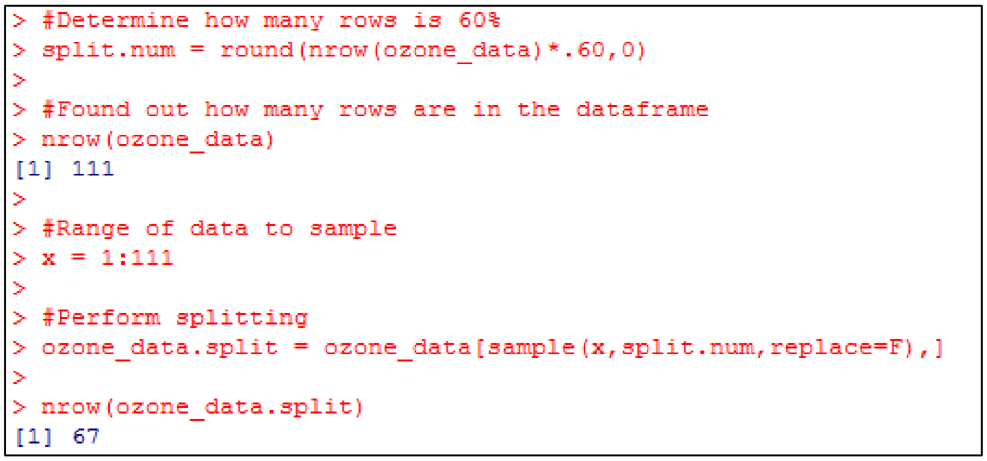
```
car_data[order(car_data$gps_bt, car_data$transmission), c(3,4)]
car_data[order(car_data$gps_bt, car_data$transmission),c("transmission","gps_bt")]
```

5. Subsampling in R

In the realms of research and analytics, many reasons exist for creating subsamples of your data. For example, you may need to create a subsample for training, testing, and validation. You may need two subsamples to perform an Exploratory Factor Analysis and a Confirmatory Factor Analysis. You may need to look at customers who are only domestic instead of international. Whatever the reason, knowing how to create subsamples is important.

A basic approach to sampling is basing your subsample on a percentage of the overall sample size. For example, say you would like to sample 60% of your original data and perform an analysis on it. The steps include

- Determine how many rows is 60% of your data
- Find out how many rows are in your dataframe
- Determine the range of your sample
- Perform the splitting

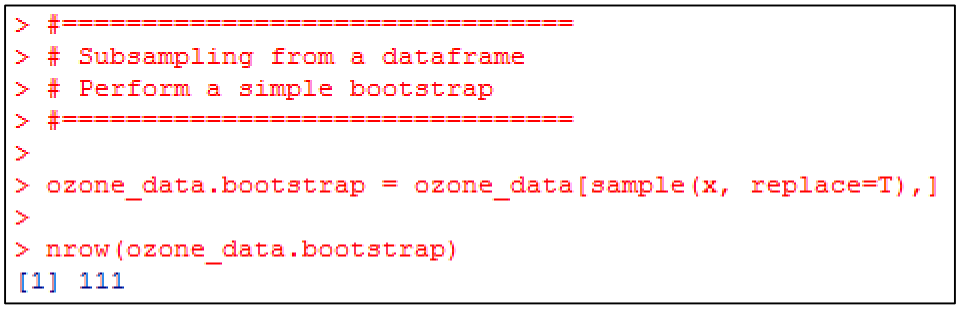


```
> #Determine how many rows is 60%
> split.num = round(nrow(ozone_data)*.60,0)
>
> #Found out how many rows are in the dataframe
> nrow(ozone_data)
[1] 111
>
> #Range of data to sample
> x = 1:111
>
> #Perform splitting
> ozone_data.split = ozone_data[sample(x,split.num,replace=F),]
>
> nrow(ozone_data.split)
[1] 67
```

Figure 8 Creating a Subsample of 60%

Look at the bottom of Figure 8. You will see the code under the comment "#Perform splitting". The "replace=F" argument means the sampling is done without replacement. This means that none of the records pulled from the data are put back into the data as the sampling progresses. Also note the value *x*. This represents the range within your dataframe that you would like to take the sample from. If you wanted to only sample from the second half of the dataset, you could have used x = 55:111 instead.

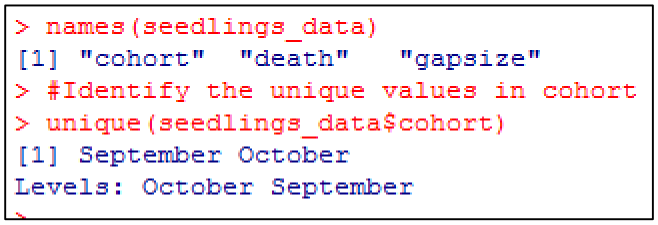
In addition to pulling actual values from a dataset, you can perform a bootstrap on your data. In the following example, the data is sampled with replacement.



```
> #================================
> # Subsampling from a dataframe
> # Perform a simple bootstrap
> #================================
>
> ozone_data.bootstrap = ozone_data[sample(x, replace=T),]
>
> nrow(ozone_data.bootstrap)
[1] 111
```

Figure 9 Bootstrap Sampling

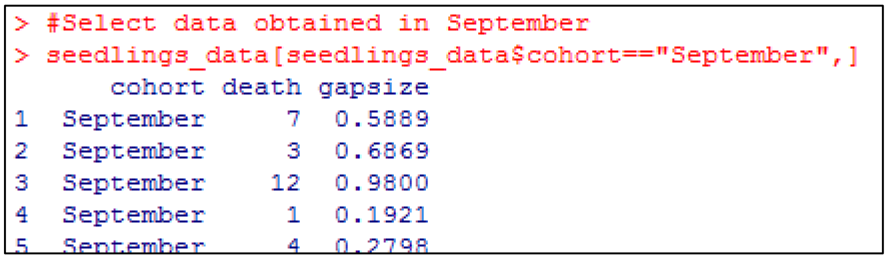
Often you will want to select a subsample based on certain conditions or criteria given the data you have. R provides a way for you to query your dataset. For this example, the seedlings_data will be used. The seedlings data contains three columns, *cohort*, *death*, and *gapsize*. *Cohort* contains two unique values as shown below.



```
> names(seedlings_data)
[1] "cohort"  "death"   "gapsize"
> #Identify the unique values in cohort
> unique(seedlings_data$cohort)
[1] September October
Levels: October September
```

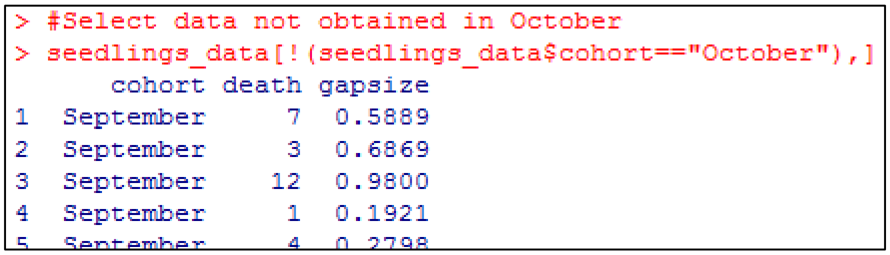
Figure 10 Seedlings Data

Assume you would like to perform an analysis only on seedlings planted in September. Or, put another way, you want data not obtained in October. The code below shows how you perform both of these operations.



```
> #Select data obtained in September
> seedlings_data[seedlings_data$cohort=="September",]
     cohort death gapsize
1  September     7  0.5889
2  September     3  0.6869
3  September    12  0.9800
4  September     1  0.1921
5  September     4  0.2798
```

Figure 11 Data Obtained in September

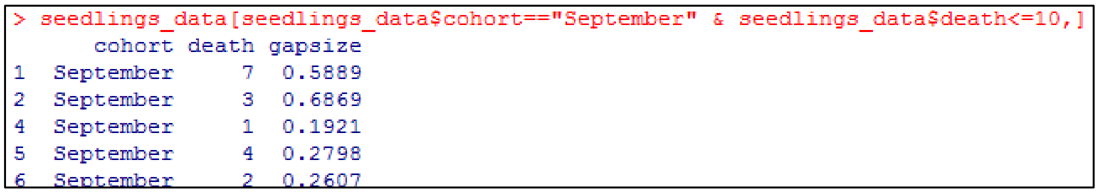


```
> #Select data not obtained in October
> seedlings_data[!(seedlings_data$cohort=="October"),]
     cohort death gapsize
1  September     7  0.5889
2  September     3  0.6869
3  September    12  0.9800
4  September     1  0.1921
5  September     4  0.2798
```

Figure 12 Data Not Obtained in October

The second example uses the exclamation mark to indicate to R that you want the opposite to happen. In this case, if the exclamation mark had been left out, you would have selected data only in October; with the exclamation mark, you want everything without October.

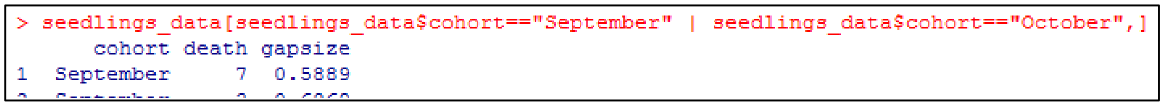
After perusing your September-data, you realize that you only want data with a death value less than or equal to 10. This is another simple process. You just append additional conditions using the symbol &.



```
> seedlings_data[seedlings_data$cohort=="September" & seedlings_data$death<=10,]
     cohort death gapsize
1  September     7  0.5889
2  September     3  0.6869
4  September     1  0.1921
5  September     4  0.2798
6  September     2  0.2607
```

Figure 13 Selection of Seedlings Based on Cohort and Death

What if you want to find data that is for September or October? You would type in both conditions and separate them using the OR operator, which is |.



```
> seedlings_data[seedlings_data$cohort=="September" | seedlings_data$cohort=="October",]
     cohort death gapsize
1  September     7  0.5889
```

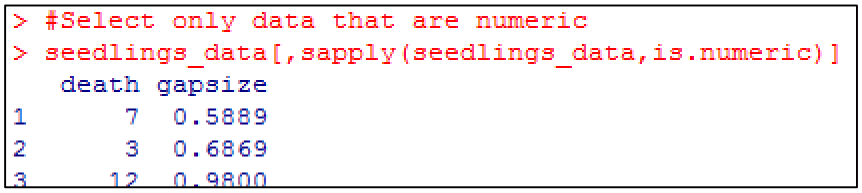
Figure 14 Using the OR Operator

Taking this one step further, you can select data for September or October and has a death value less than or equal to 10. Here are two different statements that will select two different datasets:

1. seedlings_data[(seedlings_data$cohort=="September" | seedlings_data$cohort=="October") & seedlings_data$death<=10,]
2. seedlings_data[seedlings_data$cohort=="September" | (seedlings_data$cohort=="October" & seedlings_data$death<=10),]

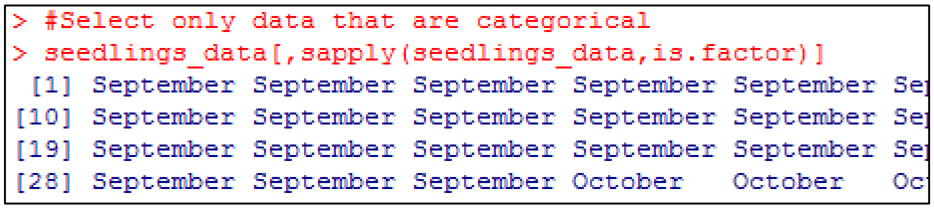
Can you spot the difference between these two? Here is a hint: Look at the placement of the parentheses. In statement 1, the parentheses surround the OR operator; in the second statement, they surround the AND operator. The first statement selects data that is 1) September and less than 10 or 2) October and less than 10. The second statement selects data that is 1) September or 2) October and less than 10. What kind of data would you select if you didn't use any parentheses? Think about that for a while.

You may want to select columns based on whether they are numeric or categorical. R provides a way to do just that.



```
> #Select only data that are numeric
> seedlings_data[,sapply(seedlings_data,is.numeric)]
   death gapsize
1      7  0.5889
2      3  0.6869
3     12  0.9800
```

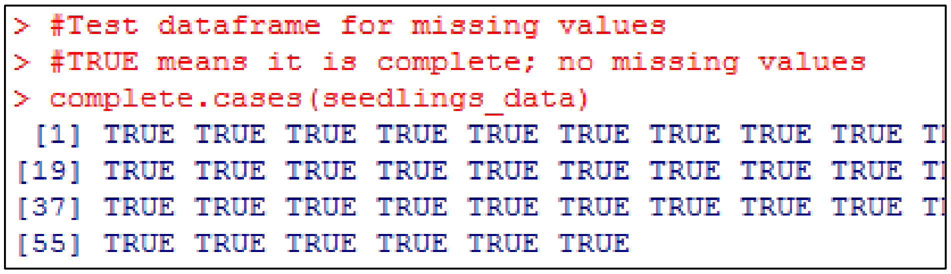
Figure 15 Selecting Numeric Columns



```
> #Select only data that are categorical
> seedlings_data[,sapply(seedlings_data,is.factor)]
 [1] September September September September September Se
[10] September September September September September Se
[19] September September September September September Se
[28] September September September October   October   Oc
```

Figure 16 Selecting Categorical Columns

You can also test your dataframe for missing values. The first method returns a TRUE-FALSE value based on whether it is complete; i.e. TRUE indicates no missing values whereas FALSE indicates missing values. It doesn't specific which column, unfortunately.



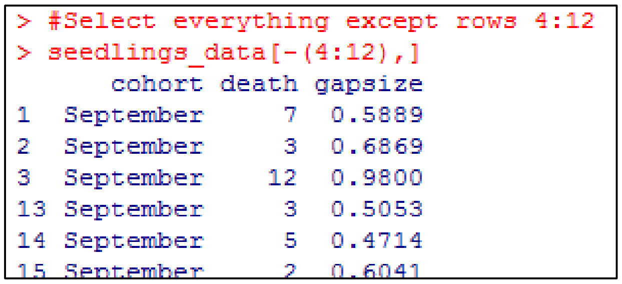
```
> #Test dataframe for missing values
> #TRUE means it is complete; no missing values
> complete.cases(seedlings_data)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE T
[19] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE T
[37] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE T
[55] TRUE TRUE TRUE TRUE TRUE TRUE
```

Figure 17 Testing for Completeness

You can also select rows that do not contain missing values and save it to a new dataframe. The R code is presented below.

seedlings_data_new = na.omit(seedlings_data)

Sometimes you would like to select data by excluding certain rows. In this example, rows 4 through 12 inclusive are left out.



```
> #Select everything except rows 4:12
> seedlings_data[-(4:12),]
      cohort death gapsize
1  September     7  0.5889
2  September     3  0.6869
3  September    12  0.9800
13 September     3  0.5053
14 September     5  0.4714
15 September     2  0.6041
```

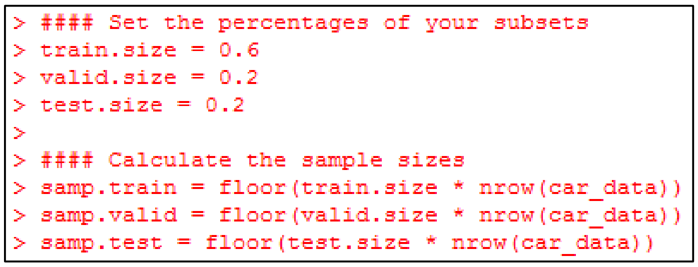
Figure 18 Leaving Out Rows

6. More on Subsampling

As an extension of the previous section on subsampling, you may wish to create subsamples to perform techniques requiring training, testing, validation data or even *k*-fold cross validation data. This is a simple extension of what was covered in the previous section. This part of the tutorial will use the car.test.frame.txt file.

Understanding how indices operate in dataframes is important for this process. Recall that the index represents the row number within a dataframe. This process randomly selects the index values from the dataframe based on the percentages for each subset.

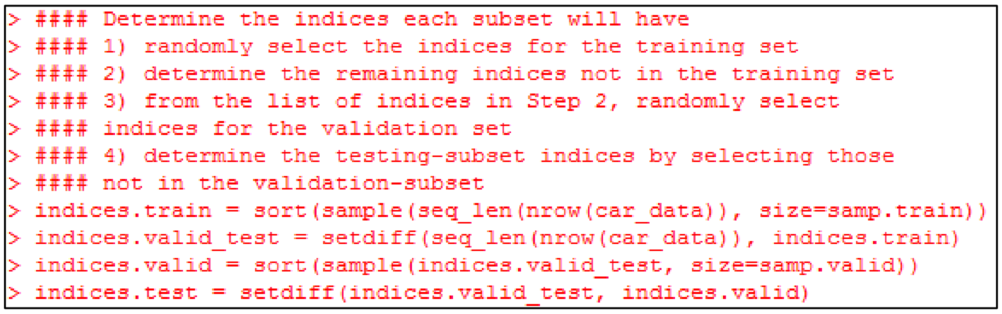
First, set the sizes, in percentages, of the subsets you are creating. Calculate the sample size for each of the subsets. See the figure below; all of the code relies on procedures you have covered.



```
> #### Set the percentages of your subsets
> train.size = 0.6
> valid.size = 0.2
> test.size = 0.2
>
> #### Calculate the sample sizes
> samp.train = floor(train.size * nrow(car_data))
> samp.valid = floor(valid.size * nrow(car_data))
> samp.test = floor(test.size * nrow(car_data))
```

Figure 19 Calculating the Sample Sizes

The second step is a bit more complicated for beginners, but the code is fairly easy to understand once the various parts have been taken apart. Again, this part of the code relies on an understanding of indices. The purpose here is to determine which rows (i.e. indices) will be placed in which subset of data. See the figure below for the code.



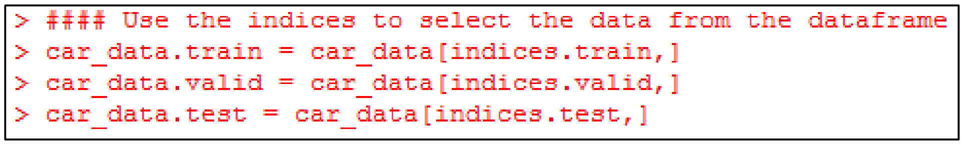
```
> #### Determine the indices each subset will have
> #### 1) randomly select the indices for the training set
> #### 2) determine the remaining indices not in the training set
> #### 3) from the list of indices in Step 2, randomly select
> #### indices for the validation set
> #### 4) determine the testing-subset indices by selecting those
> #### not in the validation-subset
> indices.train = sort(sample(seq_len(nrow(car_data)), size=samp.train))
> indices.valid_test = setdiff(seq_len(nrow(car_data)), indices.train)
> indices.valid = sort(sample(indices.valid_test, size=samp.valid))
> indices.test = setdiff(indices.valid_test, indices.valid)
```

Figure 20 Assigning Rows to Subsets

The first line of code randomly assigns indices to the training set. This is based on the sample size (60% in this example) that was chosen. The function sample() performs the random selection of indices. The data that is being fed into the function sample() is a list—or a sequence—of all the indices from the car dataset, seq_len(nrow(car_data)). The sort() function merely sorts the indices.

The second line of code determines which indices are not found in the list of indices in indices.train using the setdiff() function. The setdiff(*x*, *y*) function operates by determining which elements, items, data points, etc. in *x* are not contained in *y*. In other words, these are the indices that will be used in the validation and testing sets. The third line of code is a repeat of the first line; indices for the validation set are pulled out of the list of indices not found in the testing set. The fourth line of code uses the setdiff() function again.

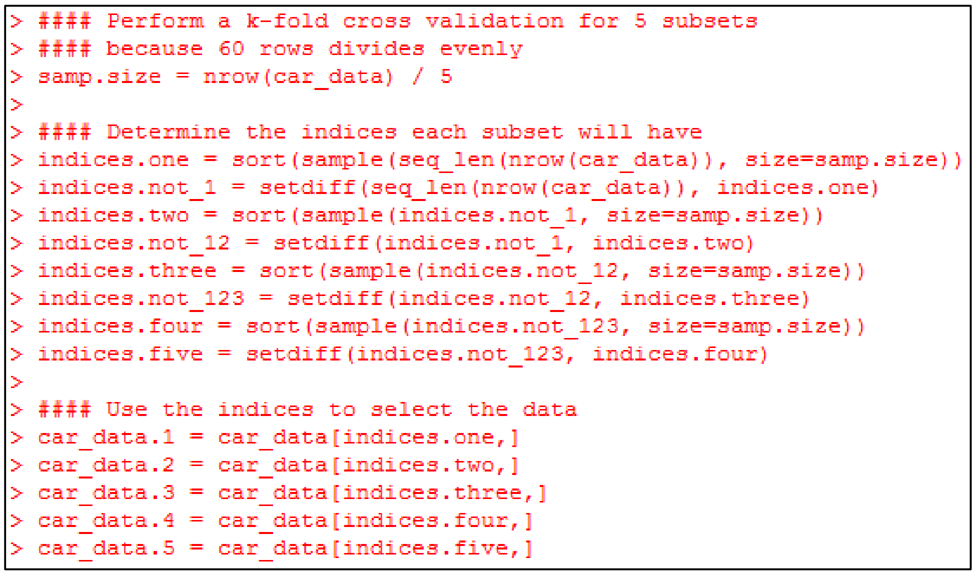
The last step is to use the three new sets of index values to pull data from car_data. See the code below. By providing a list of index values, data can be separated into multiple subsets without any overlap.



```
> #### Use the indices to select the data from the dataframe
> car_data.train = car_data[indices.train,]
> car_data.valid = car_data[indices.valid,]
> car_data.test = car_data[indices.test,]
```

Figure 21 Separate Data by Index Value

A *k*-fold cross validation approach is very similar, except each subset would have the same percentage. Additionally, the number of subsets created would not be limited to just three, as in the example given here, but up to the desired choice. An example of a 5-fold cross validation splitting of data is shown below.



```
> #### Perform a k-fold cross validation for 5 subsets
> #### because 60 rows divides evenly
> samp.size = nrow(car_data) / 5
>
> #### Determine the indices each subset will have
> indices.one = sort(sample(seq_len(nrow(car_data)), size=samp.size))
> indices.not_1 = setdiff(seq_len(nrow(car_data)), indices.one)
> indices.two = sort(sample(indices.not_1, size=samp.size))
> indices.not_12 = setdiff(indices.not_1, indices.two)
> indices.three = sort(sample(indices.not_12, size=samp.size))
> indices.not_123 = setdiff(indices.not_12, indices.three)
> indices.four = sort(sample(indices.not_123, size=samp.size))
> indices.five = setdiff(indices.not_123, indices.four)
>
> #### Use the indices to select the data
> car_data.1 = car_data[indices.one,]
> car_data.2 = car_data[indices.two,]
> car_data.3 = car_data[indices.three,]
> car_data.4 = car_data[indices.four,]
> car_data.5 = car_data[indices.five,]
```

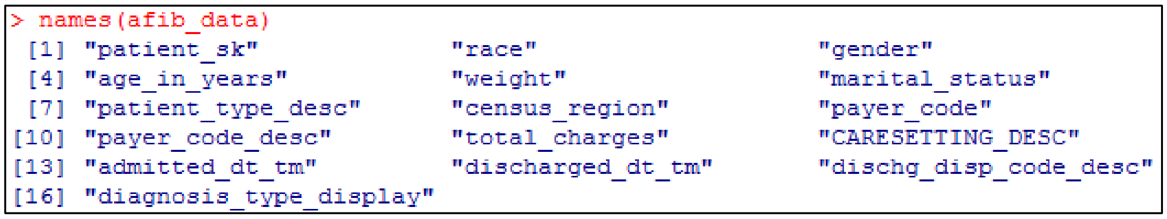
Figure 22 5-Fold Cross Validation Data

7. Dates and Times in R

All modern statistical packages provide functions that perform mathematical operations and dates and times. For example, say you have data on employee work hours and you need to calculate pay for hourly employees. For each day over a five-day span you have the time the employee clocked in and the time the employee clocked out. You need to calculate the total number of hours the employee worked by using one of two methods: 1) you convert the date-time values into hours-minutes-seconds and sum up the values or 2) use a date-time function that will automatically convert for you and provide you the total hours.

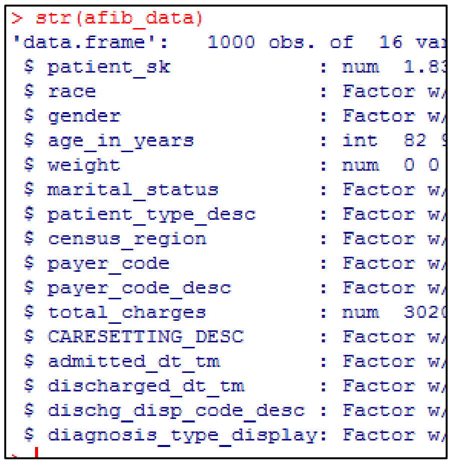
The second option is the obvious choice as it requires minimal computational skills on your part. R has a function, difftime(), that will provide the difference between two date-time values. One of the downsides to most statistical packages is they do not convert date-time values into date-time objects. That is, date-time values are read as categorical values made up of character strings; you cannot perform math on character strings. This means that functions like difftime() will not work on the data.

Converting date-time values in any statistical program requires work, and R is no exception. For this portion of the tutorial you will be given an example of how to convert date-time values from categorical datatypes to date-time datatypes. Use the afib_data.txt file to follow along. The figure below contains the column names of the dataset. This contains data on atrial fibrillation patients.



```
> names(afib_data)
 [1] "patient_sk"          "race"                    "gender"
 [4] "age_in_years"        "weight"                  "marital_status"
 [7] "patient_type_desc"   "census_region"           "payer_code"
[10] "payer_code_desc"     "total_charges"           "CARESETTING_DESC"
[13] "admitted_dt_tm"      "discharged_dt_tm"        "dischg_disp_code_desc"
[16] "diagnosis_type_display"
```

Figure 23 Columns in the Atrial Fibrillation Data

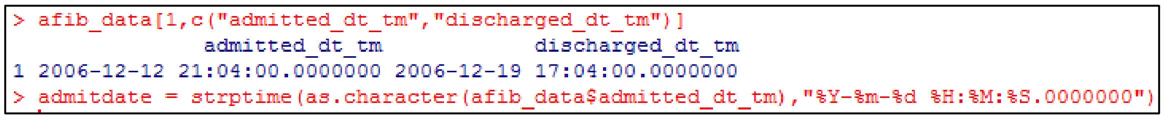


```
> str(afib_data)
'data.frame':   1000 obs. of  16 va
 $ patient_sk           : num  1.83
 $ race                 : Factor w/
 $ gender               : Factor w/
 $ age_in_years         : int  82 9
 $ weight               : num  0 0
 $ marital_status       : Factor w/
 $ patient_type_desc    : Factor w/
 $ census_region        : Factor w/
 $ payer_code           : Factor w/
 $ payer_code_desc      : Factor w/
 $ total_charges        : num  302(
 $ CARESETTING_DESC     : Factor w/
 $ admitted_dt_tm       : Factor w/
 $ discharged_dt_tm     : Factor w/
 $ dischg_disp_code_desc : Factor w/
 $ diagnosis_type_display: Factor w/
```

Figure 24 Datatypes for A-Fib Data

As mentioned previously, R imports date-time values as categorical datatypes. The dataset contains two date-time columns: admitted_dt_tm and discharged_dt_tm. Looking at Figure 24, the datatypes for both are listed as "Factor" or categorical. The next step is to convert these two columns into a date-time datatype.

The function in R to convert a string into a date-time datatype is strptime() or "strip time." This function is dependent on your ability to format the sequence of numbers and delimiters correctly in order to convert it to a date-time datatype. The first line of Figure 25 shows how the date-time is formatted in the dataset. Notice that it has the year (in four digits) first, then a dash, then the month, then another dash, and then the day. After this a space follows, and then the hour (a 24-hour format), the minute, the seconds, and seven trailing zeroes.



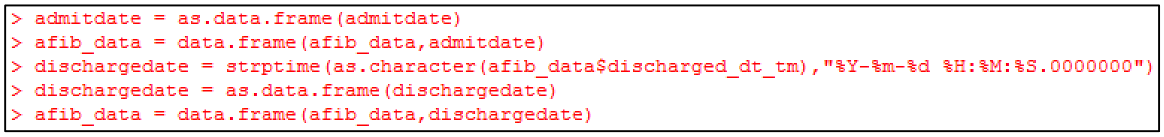
```
> afib_data[1,c("admitted_dt_tm","discharged_dt_tm")]
             admitted_dt_tm          discharged_dt_tm
1 2006-12-12 21:04:00.0000000 2006-12-19 17:04:00.0000000
> admitdate = strptime(as.character(afib_data$admitted_dt_tm),"%Y-%m-%d %H:%M:%S.0000000")
```

Figure 25 Converting Patient Admit Date into Date-Time Datatype

The second line of the output uses the strptime() function. The first part that uses the as.character() function merely converts the admitted_dt_tm column into a character string in case it isn't already. The next part tells R what the format of the date-time character string is. The %Y represents a four-digit year, as opposed to %y that represent a two-digit year. The %m represents a two-digit month. Notice a dash lies between %Y and %m. This is because the actual data has a dash. If the year and month had a forward slash, like “2016/05/23,” then you would have to type “%Y/%m/%d.” The following are all of the formatting characters used by R for date-time datatypes:

- %a Abbreviated weekday name
- %A Full weekday name
- %b Abbreviated month name
- %B Full month name
- %c Date and time, locale-specific
- %d Day of the month as decimal number (01–31)
- %H Hours as decimal number (00–23) on the 24-hour clock
- %I Hours as decimal number (01–12) on the 12-hour clock
- %j Day of year as decimal number (001–366)
- %m Month as decimal number (01–12)
- %M Minute as decimal number (00–59)
- %p AM/PM indicator in the locale
- %S Second as decimal number (00–61, allowing for two ‘leap seconds’)
- %U Week of the year (00–53) using the first Sunday as day 1 of week 1
- %w Weekday as decimal number (0–6, Sunday is 0)
- %W Week of the year (00–53) using the first Monday as day 1 of week 1
- %x Date, locale-specific
- %X Time, locale-specific
- %Y Year with century
- %Z Time zone as a character string (output only)

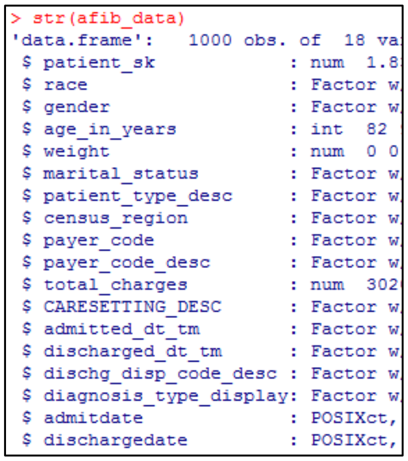
Now that the column admitted_dt_tm is converted to a date-time datatype, it needs to be placed back into the original dataframe. To do this, it must first be converted into a dataframe and then appended to the end of the original dataframe. The object admitdate is just a vector and cannot be placed into the dataframe, so it must be converted to a dataframe. The following code shows how this is done, along with the conversion of the column discharged_dt_tm.



```
> admitdate = as.data.frame(admitdate)
> afib_data = data.frame(afib_data,admitdate)
> dischargedate = strptime(as.character(afib_data$discharged_dt_tm),"%Y-%m-%d %H:%M:%S.0000000")
> dischargedate = as.data.frame(dischargedate)
> afib_data = data.frame(afib_data,dischargedate)
```

Figure 26 Adding Admitdate Back into the Original Dataframe

Looking at Figure 27 to the right, the last two columns listed show a datatype of POSIXct. This is one of the date-time datatypes found in R. The “ct” stands for *continuous time*. The other date-time datatype is POSIXlt, where “lt” stand for *list time*. You can convert back and forth between the two. When performing date-time operations, ensure that your columns of data are all POSIXct or all POSIXlt, not both.



```
> str(afib_data)
'data.frame':   1000 obs. of  18 va
 $ patient_sk            : num  1.8
 $ race                  : Factor w
 $ gender                : Factor w
 $ age_in_years          : int  82
 $ weight                : num  0 0
 $ marital_status        : Factor w
 $ patient_type_desc     : Factor w
 $ census_region         : Factor w
 $ payer_code            : Factor w
 $ payer_code_desc       : Factor w
 $ total_charges         : num  302
 $ CARESETTING_DESC      : Factor w
 $ admitted_dt_tm        : Factor w
 $ discharged_dt_tm      : Factor w
 $ dischg_disp_code_desc : Factor w
 $ diagnosis_type_display: Factor w
 $ admitdate             : POSIXct,
 $ dischargedate         : POSIXct,
```

Figure 27 Datatypes for Dataframe