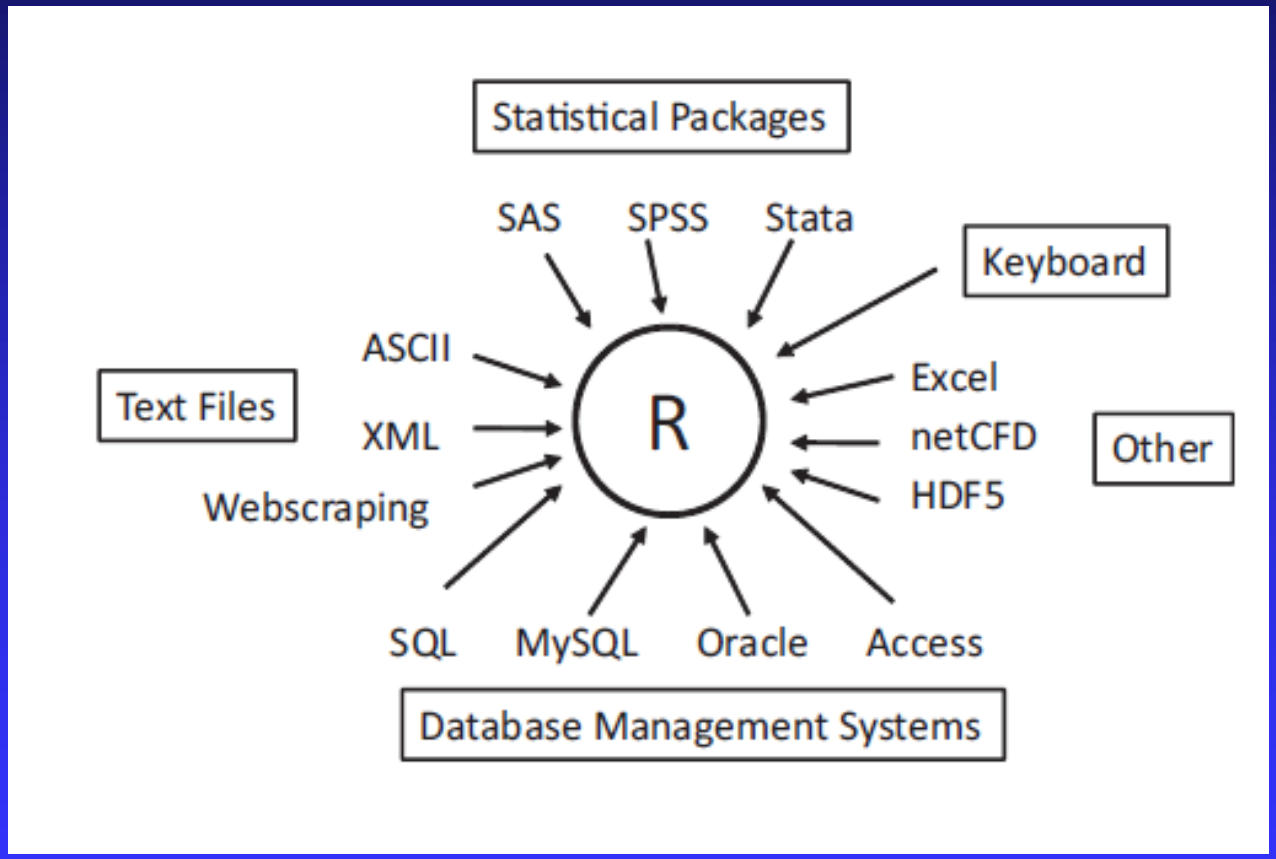
# Lecture 4

## Managing Data with R

# Sources of data that can be imported into R



# Saving and loading R objects

An R object can be saved with the *save()* function. For example, the following command saves pt_data to the file pt_data.RData:

```
> save(pt_data,file="pt_data.RData")
```

By default, the file is in your Windows Document folder. Now, you can load your object back into R with the load function:

```
> load("pt_data.RData")
```

Loading an object from a file will overwrite an existing object with the same name in the current session. You can also provide an absolute path to the file:

```
> save(pt_data,file="c:/Documents and
 +   Settings/svet/pt_data.RData")
```

In R, paths are always specified with forward slashes (“/”) even on Microsoft Windows. Also note that the file argument must be explicitly named.

You can save multiple objects in the same file with one save command:

```
> save(z,v,pt_data, file="pt_data.RData")
```

If you want to save every object in your workplace/session, you can use the *save.image()* function

# Saving and loading R objects *(cont.)*

*The ls()* listing function returns a vector of all the data structures currently in the memory. The *rm()* function removes an object from the working memory. For example,

```
> rm(x, y, z)
```

removes x, y and z from the working memory.

*rm()* can be combined with the *ls()* function to clear the entire R session:

```
> rm(list=ls())
```

# Editing data

The edit() function allows an object to be edited with data editor:

```
> pt_data1 <- edit(pt_data)
```

This opens the Data Editor window shown on the next slide. After modifying pt_data, the result is saved into pt_data1. The following command stores the changes back in pt_data

```
> pt_data <- edit(pt_data)
```

The *fix()* function command is very similar to the *edit()* function. The only difference is that the *fix()* function calls edit on its argument and assigns the result to the same name. In other words,

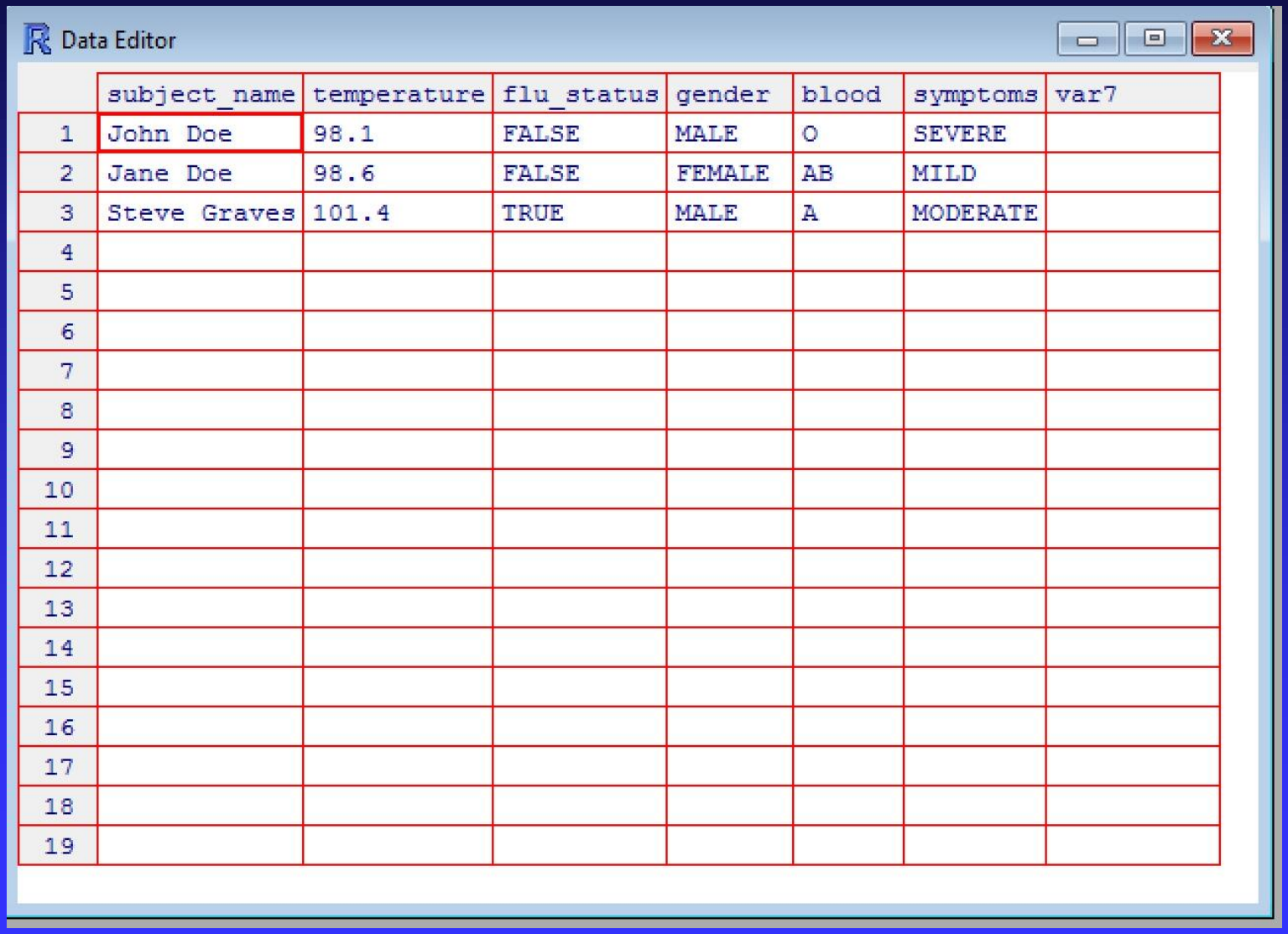
```
> fix(pt_data)
```

Is equivalent to:

```
> pt_data <- edit(pt_data)
```

You have to be careful: the Data Editor does not provide Undo or Redo functions. Second, there is no Save button. You have to close the editor and save your work.

# The Data Editor window




| | subject_name | temperature | flu_status | gender | blood | symptoms | var7 |
|---|---|---|---|---|---|---|---|
| 1 | John Doe | 98.1 | FALSE | MALE | O | SEVERE | |
| 2 | Jane Doe | 98.6 | FALSE | FEMALE | AB | MILD | |
| 3 | Steve Graves | 101.4 | TRUE | MALE | A | MODERATE | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | | | | |
| 7 | | | | | | | |
| 8 | | | | | | | |
| 9 | | | | | | | |
| 10 | | | | | | | |
| 11 | | | | | | | |
| 12 | | | | | | | |
| 13 | | | | | | | |
| 14 | | | | | | | |
| 15 | | | | | | | |
| 16 | | | | | | | |
| 17 | | | | | | | |
| 18 | | | | | | | |
| 19 | | | | | | | |

# Importing data from text files

R can import data from text files, other statistics software, and even spreadsheets.

Most text files containing data are formatted in a **tabular form:** each line of a text file represents an observation (or example or record) and it contains a set of different variables/features  associated with that observation.

The feature values on each line are separated by a predefined symbol, known as a **delimiter**. Often, the first line of a tabular data file lists the names of the columns of data. This is called a **header** line.

The most versatile function for importing delimited files is the read.table() function. Here is the general format of the function:

read.table(file, header, sep = , quote = , dec = , row.names, col.names,

as.is = , na.strings , colClasses , nrows =, skip = ,

check.names = , fill = , strip.white = , blank.lines.skip = ,

comment.char = , allowEscapes = , flush = , stringsAsFactors = ,

encoding = )

A brief description of the function arguments is provided on the next side.

# Arguments of read.table()

| Argument | Description | Default |
|---|---|---|
| file | The name of the file to open or, alternatively, the name of a connection containing the data. You can even use a URL. (This is the one required argument for `read.table`.) | |
| header | A logical value indicating whether the first row of the file contains variable names. | `FALSE` |
| sep | The character (or characters) separating fields. When “” is specified, any white space is used as a separator. | `""` |
| quote | If character values are enclosed in quotes, this argument should specify the type of quotes. | “” |
| dec | The character used for decimal points. | `.` |
| row.names | A character vector containing row names for the returned data frame. | |
| col.names | A character vector containing column names for the returned data frame. | |
| as.is | A logical vector (the same length as the number of columns) that specifies whether or not to convert character values to factors. | `!stringsAsFactors` |
| na.strings | A character vector specifying values that should be interpreted as NA. | NA |
| colClasses | A character vector of class names to be assigned to each column. | NA |
| nrows | An integer value specifying the number of rows to read. (Invalid values, such as negatives, are ignored.) | `-1` |

# Arguments of read.table()

| Argument | Description | Default |
|---|---|---|
| skip | An integer value specifying the number of rows in the text file to skip before beginning to read data. | 0 |
| check.names | A logical value that specifies whether `read.table` should check if the column names are valid symbol names in R. | TRUE |
| fill | Sometimes, a file might contain rows of unequal length. This argument is a logical value that specifies whether `read.table` should implicitly add blank fields at the end of rows where some values were missing. | `!blank.lines.skip` |
| strip.white | When `sep !=""`, this logical value specifies whether `read.table` should remove extra leading and trailing white space from character fields. | FALSE |
| blank.lines.skip | A logical value that specifies whether `read.table` should ignore blank lines. | TRUE |
| comment.char | `read.table` can ignore comment lines in input files if the comment lines begin with a single special character. This argument specifies the character used to delineate these lines. | `"#"` |
| allowEscapes | A logical value that indicates whether escapes (such as "\n" for a new line) should be interpreted or if character strings should be read literally. | FALSE |
| flush | A logical value that indicates whether `read.table` should skip to the next line when all requested fields have been read in from a line. | FALSE |
| stringsAsFactors | A logical value indicating whether text fields should be converted to factors. | `default.stringsAsFactors()` |
| encoding | The encoding scheme used for the source file. | `"unknown"` |

# Importing data from text files *(cont.)*

For example, suppose that we have a file called pt_data.txt that contains the following text:

```
"subject_name",   "temperature", "flu_status", "gender", "blood", "symptoms"
"1","John Doe",    98.1,          FALSE,       "MALE",    "O",     "SEVERE"
"2","Jane Doe",    98.6,          FALSE,       "FEMALE",  "AB",    "MILD"
"3","Steve Graves",101.4,         TRUE,        "MALE",    "A",     "MODERATE"
```

The file contains the same data we have entered into pt_data data frame. The format is tabular. The first row contains the column names; each text field is encapsulated in quotes; and fields are separated by commas. To load this file into R, you must specify that the first row contains column names (header=TRUE), that the delimiter is comma (sep=","), and that quotes are used to encapsulate text (quote="\""). Here is an R statement that loads the file:

```
> pt_data <-read.table("pt_data.txt", quote="\"", sep=",",
+ header=TRUE)
```

# Importing data from text files *(cont.)*

R includes a set of built-in functions that call read.table() with different default options. Here is a table with the functions and their options.

| Function | header | sep | quote | dec | fill | comment.char |
|---|---|---|---|---|---|---|
| read.table | FALSE | | \" or \' | . | ! blank.lines.skip | # |
| read.csv | TRUE | , | \" | . | TRUE | |
| read.csv2 | TRUE | ; | \" | , | TRUE | |
| read.delim | TRUE | \t | \" | . | TRUE | |
| read.delim2 | TRUE | \t | \" | , | TRUE | |

# Importing data from text files *(cont.)*

One of the most common tabular text file format is the **CSV** (**Comma-Separated Values**), which uses the comma as a delimiter. CSV files can be imported using the read.csv() function. Suppose we have the following data stored in pt_data.csv:

subject_name, temperature, flu_status,gender, blood_type

John Doe,98.1,FALSE,MALE,O

Jane Doe,98.6,FALSE,FEMALE,AB

Steve Graves,101.4,TRUE,MALE,A

The read.csv() function can be used to load the file into R:

```
> pt_data <- read.csv("pt_data.csv", stringsAsFactors =
+ FALSE)
```

The option stringsAsFactors = FALSE was used to prevent R from converting all text variables into factors.

The read.csv() function assumes that the CSV file includes a header line listing the names of the features in the dataset. If a CSV file does not have a header, we need to use the option header = FALSE.

# Exporting Data

R can export R data objects (usually data frames and matrices) as text files. To export data to a text file, use the write.table function:

write.table(x, file = "", append = FALSE, quote = TRUE, sep = " ", eol = "\n", na = NA", dec = ".", row.names = TRUE, col.names = TRUE, qmethod = c("escape", "double"))

A brief description of the arguments is provided on the next slide.

There are built-in functions that call *write.table(…)* with different defaults such as write.csv(...), write.csv2(...), etc. For example. You can save pt_data into a CSV file using the following command:

```
> write.csv(pt_data, file="pt_data1.csv", eol="\n")
```

# Arguments to write.table

| Argument | Description | Default |
|---|---|---|
| x | Object to export. | |
| file | Character value specifying a filename or a connection object to which you would like to write the output. | "" |
| append | A logical value indicating whether to append the output to the end of an existing file (append=TRUE) or replace the file (append=FALSE). | FALSE |
| quote | A logical value specifying whether to surround any character or factor values with quotes, or a numeric vector specifying which columns to surround with quotes. | TRUE |
| sep | A character value specifying the value that separates values within a row. | "" |
| eol | A character value specifying the value to append on the end of each line. | "\n" |
| na | A character value specifying how to represent NA values. | "NA" |
| dec | A character value specifying the decimal separator in numeric values. | "." |
| row.names | A logical value indicating whether to include row names in the output or a numeric vector specifying the rows from which row names should be included. | TRUE |
| col.names | A logical value specifying whether to include column names or a character vector specifying alternate names to include. | TRUE |
| qmethod | Specifies how to deal with quotes inside quoted character and factor fields. Specify qmethod="escape" to escape quotes with a backslash (as in C) or qmethod="double" to escape quotes as double quotes (i.e., " is transformed to ""). | "escape" |

# Exploring and understanding data

# Exploring the structure of data

Download the usedcars.csv dataset from the example code of Chapter 2 (all example code is posted in the Course Materials Section on Blackboard) Then, you need the *read.csv()* function to load the data into an R data frame:

```
> usedcars <- read.csv("usedcars.csv", stringsAsFactors =
FALSE)
```

Since we do not know what the structure of the data is, we can run the *str()* function which displays a summary of the data structure:

```
> str(usedcars)
'data.frame':   150 obs. of  6 variables:
 $ year        : int  2011 2011 2011 2011 2012 2010 2011 2010 2011 2010 ...
 $ model       : chr  "SEL" "SEL" "SEL" "SEL" ...
 $ price       : int  21992 20995 19995 17809 17500 17495 17000 16995 16995 16995
...
 $ mileage     : int  7413 10926 7351 11613 8367 25125 27393 21026 32655 36116 ...
 $ color       : chr  "Yellow" "Gray" "Silver" "Gray" ...
 $ transmission: chr  "AUTO" "AUTO" "AUTO" "AUTO" ...
```

# Exploring the structure of data *(cont.)*

Since the data frame is small, a brief overview of the six variables/columns can be gained by running directly:

```
> usedcars
     year model price mileage  color transmission
1    2011   SEL 21992    7413 Yellow         AUTO
2    2011   SEL 20995   10926   Gray         AUTO
3    2011   SEL 19995    7351 Silver         AUTO
4    2011   SEL 17809   11613   Gray         AUTO
5    2012    SE 17500    8367  White         AUTO
6    2010   SEL 17495   25125 Silver         AUTO
7    2011   SEL 17000   27393   Blue         AUTO
8    2010   SEL 16995   21026 Silver         AUTO
9    2011   SES 16995   32655 Silver         AUTO
.....
.....
```

# Exploring the structure of data *(cont.)*

```
head(usedcars)          # shows the first 6 rows of the dataset
head(usedcars, n=10)    # shows the first 10 rows of the dataset
head(usedcars, n=-10)   # shows all but the last 10 rows of the dataset
tail(usedcars)          # last 6 rows
tail(usedcars, n=10)    # last 10 rows
tail(usedcars, n= -10)  # all rows but the first 10
names(usedcars)         # lists variables in the dataset
```

# Exploring the structure of data *(cont.)*

*str()* shows all the elements of an object and then recursively shows each element's contents and attributes. Long vectors and lists are truncated to keep the output manageable. We can see that we have loaded a data frame. The statement 150 obs says that the data includes 150 observations, i.e. 150 car records. Every record includes the values of six variables (or features): year, model, price, mileage, color, and transmission. "chr" and "int" labels tell us the type of each variable: character or integer. After each variable's type, there is a sequence of the first few feature values. The values "Yellow" "Gray" "Silver" "Gray" are the first four values of the color feature.

To explore the numeric nature of data we can run summary statistics for integer variables. For example:

```
> summary(usedcars$year)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2000    2008    2009    2009    2010    2012
```

You can also use the *summary()* function to obtain summary statistics for several numeric variables at the same time:

```
> summary(usedcars[c("price", "mileage")])
     price          mileage
 Min.   : 3800   Min.   :  4867
 1st Qu.:10995   1st Qu.: 27200
 Median :13592   Median : 36385
 Mean   :12962   Mean   : 44261
 3rd Qu.:14904   3rd Qu.: 55125
 Max.   :21992   Max.   :151479
```

# Quartiles

The *summary()* function shows the minimum value, the first quartile, the median, the third quartile and the maximum value. By definition, the first quartile is the middle number between the minimum number and the median of the data set. The second quartile  is the median of the data. The third quartile is the middle value between the median and the maximum value of the data set. As a result, the quartiles of a an ordered set of data values are the three points that divide the data set into four equal groups, each group comprising a quarter of the data. The second quartile is always the median.

# Finding the range

As you remember the range of a sample is a measure of the sample's spread. In R, the *range()* function returns a vector containing both the minimum and maximum value:

```
> range(usedcars$price)
```

[1] 3800 21992

Alternatively

```
> max(usedcars$price)-min(usedcars$price)
[1] 18192
```

Hence, the range is 18192. Instead of directly subtracting 3800 from 21992 to find the range, you can run the difference function:

```
> diff(range(usedcars$price))
[1] 18192
```

In the most typical case, the range function takes a vector or a matrix and returns a vector or a matrix of the differences between consecutive elements. For example:

```
> diff(c(1,3,8))
[1] 2 5
```

The order of the elements matters:

```
> diff(c(8,3,1))
[1] -5 -2
```

Similarly, you can find the range of the year and mileage variables:

```
> diff(range(usedcars$year))
[1] 12
> diff(range(usedcars$mileage))
[1] 146612
```

# Finding eventual outliers

As you remember, The difference between Q1 and Q3 is known as the **Interquartile Range** (**IQR**), and it can be calculated with the IQR() function:

```
> IQR <- IQR(usedcars$price)
> IQR
[1] 3909.5
```

We can use IQR to find eventual price outliers:

```
> summary(usedcars$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 3800   11000   13590   12960   14900   21990
> left <- 11000-1.5*IQR
> right <- 14900+1.5*IQR
> left
[1] 5135.75
> right
[1] 20764.25
> x <- usedcars$price
> x[x<left | x>right]
   [1] 21992 20995  4899  3800
```

In other words, we have two cars with very high prices and two cars with very low prices.

# Finding eventual outliers *(cont.)*

Let's find the car with the lowest price of $3800. First, we create a logical vector, which checks if the price of each car is equal to 3800.

```
> L <- usedcars$price==3800
```

Then, we use L as an index vector to extract the car records with price equal to 3800:

```
> usedcars[L,]      # we extract all columns for every row
  year model price mileage  color transmission
50 2000    SE  3800  109259   Red         AUTO
```

One reason for the low price could be the relatively high mileage and the fact that the car is relatively old. The range of years is between 2000 and 2012:

```
> range(usedcars$year)
[1] 2000 2012
```

Let's check if this car is also an outlier in terms of mileage:

```
> IQR1 <- IQR(usedcars$mileage)
> IQR1
[1] 27924.25
```

# Finding eventual outliers *(cont.)*

```
> summary(usedcars$mileage)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   4867   27200   36380   44260   55120  151500
> left1 <- 27200 - IQR1*1.5
> right1 <- 55120 + IQR1*1.5
> x <- usedcars$price
> y <- usedcars$mileage
> y[y<left1 | y>right1]
[1] 127327 106171  97987 105714 101130 119720 151479 109259
```

Yes, this particular car with a mileage of 109259 is among the mileage outliers, i.e., the car is both price and mileage outlier.

If you run the quantile() function with increments of 20%, you will see that the price is not uniformly distributed:

```
> quantile(usedcars$price, seq(from = 0, to = 1, by = 0.20))
     0%     20%     40%     60%     80%    100%
 3800.0 10759.4 12993.8 13992.0 14999.0 21992.0
```

That is, 80% of the cars have prices higher than $10759.

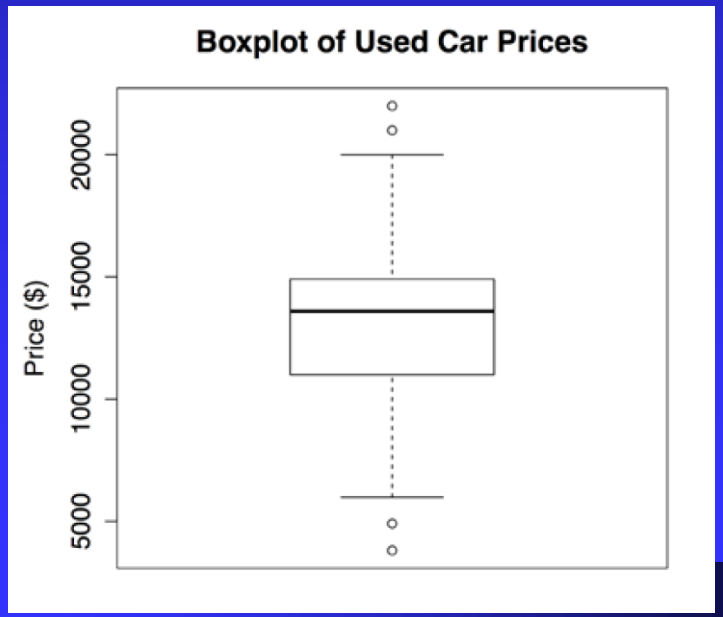
# Boxplots

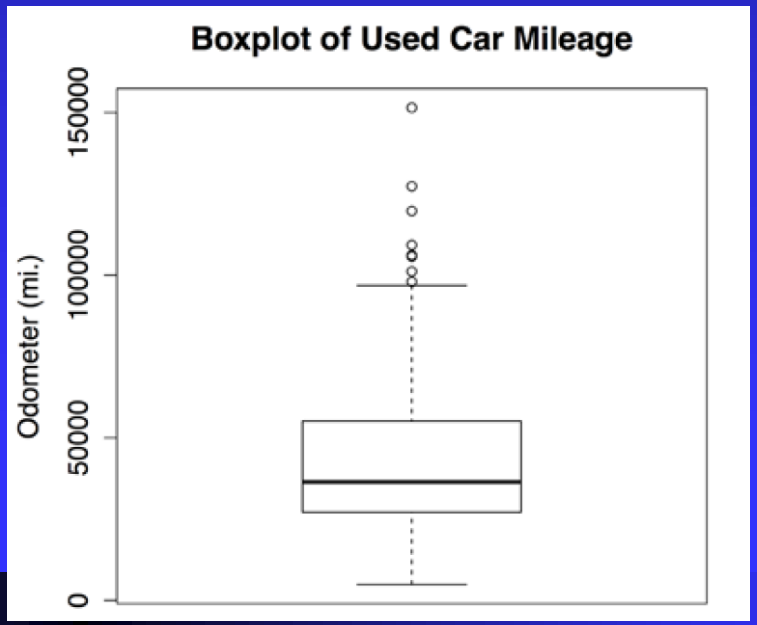
Let's look at a boxplot for the used car price and mileage.

```
> boxplot(usedcars$price, main="Boxplot of Used Car Prices",
 + ylab="Price ($)")
> boxplot(usedcars$mileage, main="Boxplot of Used Car Mileage",
 + ylab="Odometer (mi.)")
```

The horizontal lines forming the box in the middle of each figure represent Q1, Q2 (the median), and Q3 while reading the plot from the bottom to the top. The median is denoted by the dark line, which lines up with $13,592 on the vertical axis for price and 36,385 mi. on the vertical axis for mileage. Outliers and are denoted as circles (two outliers on the high end for price, two outliers on the low end for price, and several outliers on the high end for mileage.

The boxplots show that they are cars which are bot price and mileage outliers.



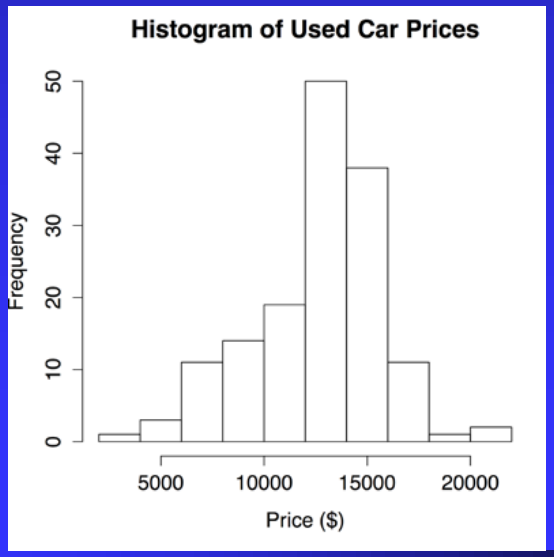



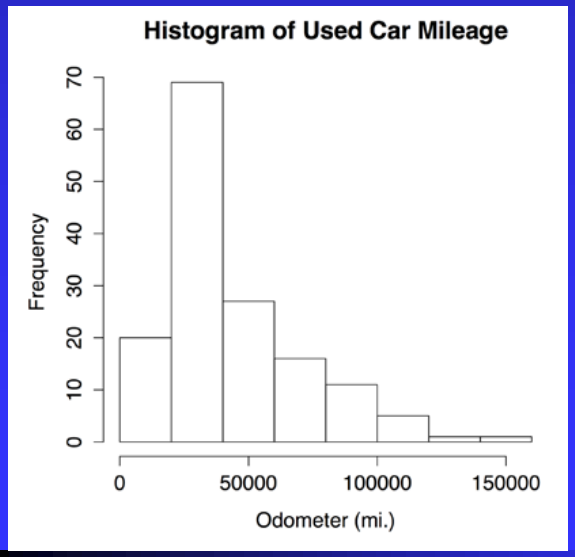
# Histogram

Histograms are another way to visualize numeric data. Histograms can be created using the following commands:

```
> hist(usedcars$price, main = "Histogram of Used Car Prices", xlab =
+  "Price ($)")
> hist(usedcars$mileage, main = "Histogram of Used Car Mileage", xlab
+  = "Odometer (mi.)")
```

The *main* attribute is used to set the title of the histogram and the *xlab* attribute is used to label the x axis. The histogram is composed of a series of bars with heights indicating the frequency of values falling within each of the equal width bins partitioning the values.



# Histogram *(cont.)*

The tallest bar at the center of the price histogram covers the $12,000 to $14,000 range and has a frequency of about 50 (49 to be precise). Since the data frame includes 150 car records, we can conclude that one-third of all the cars are priced from $12,000 to $14,000.

In the mileage histogram, the tallest bar is not at the center of the data, but on the left-hand side of the diagram. It includes about 70 cars (69 to be precise) with odometer readings from 20,000 to 40,000 miles.

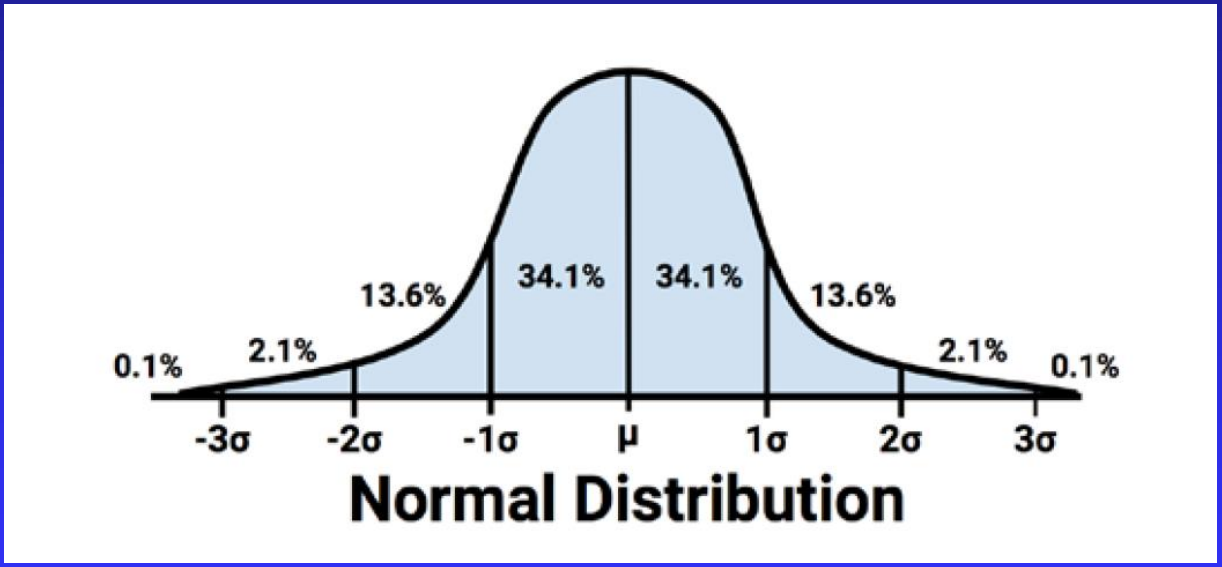
To find the exact number of cars priced between $12,000 and $14,000, we can run the following command:

```
> length(which(x>12000 & x<14000))
[1] 49
```

That is, there are 49 cars with prices between $12,000 and $14,000. The expression x>12000 & x<14000 creates a logical vector of length 150. The TRUE values of the vector correspond to the elements that satisfy the logical condition. The function *which()* returns the numeric indices of the TRUE values. The *length()* function returns the length of a vector.

# The Empirical (Normal) rule

According to the rule, 68 percent of the values in a normal distribution fall within one standard deviation of the mean, while 95 percent and 99.7 percent of the values fall within two and three standard deviations, respectively. This is illustrated in the following diagram:




Let’s check if the price distribution is close to normal.

# The Empirical (Normal) rule *(cont.)*

```
> meanPrice <- mean(x) # the price mean is stored into meanPrice
> sdPrice <- sd(x)              # the price standard  deviation is
                                # stored into sdPrice
> length(which(x > meanPrice - sdPrice & x < meanPrice + sdPrice))
[1] 113                 # count the number of data values which are one
                        #  standard deviation from the mean
                        # this is about 75%
> length(which(x > meanPrice - 2*sdPrice & x < meanPrice + 2*sdPrice))
[1] 142                  # count the number of data values which are two
                        # standard deviations from the mean
                        # this is about 95%
> length(which(x > meanPrice - 3*sdPrice & x < meanPrice + 3*sdPrice))
[1] 150                 # count the number of data values which are
                        # three standard deviations from the mean
                        # this is 100%
```

We can conclude that the price distribution is close to the normal distribution.

According to the histogram, the mileage distribution is right-skewed.

# Finding the mode

As we know, the mode is the value that occurs most often in the data set. Although the mode is the most frequent value, it is not necessarily a majority of value, i.e., the frequency of the mode could be less than 50%.

Here are two commands that can help you find the mode of car prices:

```
> temp <- table(as.vector(x))  # x is usedcars$prices
> names(temp)[temp == max(temp)]
[1] "12995"
```

That is, the most frequent price is %12,995.

The first command temp <- table(as.vector(x)) creates a table called "temp“ with two rows. The first row of the table is a sorted list of all unique values in the vector x. The second row of temp counts the occurrences of each unique value. For example, if you run:

```
> temp
```

You will get the temp table containing different prices and their frequencies (shown on the next slide).

The second command, `names(temp)[temp == max(temp)]`, returns the most frequent price, i.e., the mode.

# Finding the mode *(cont.)*

| 3800 | 4899 | 5980 | 5995 | 6200 | 6950 | 6980 | 6995 | 6999 | 7488 | 7900 | 7995 | 7999 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 | 1 |

| 8480 | 8494 | 8495 | 8800 | 8996 | 8999 | 9000 | 9651 | 9992 | 9995 | 9999 | 10000 | 10717 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 |

| 10770 | 10815 | 10836 | 10955 | 10979 | 10995 | 11450 | 11495 | 11749 | 11754 | 11792 | 11980 | 11984 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| 11992 | 11999 | 12280 | 12500 | 12507 | 12595 | 12704 | 12777 | 12780 | 12849 | 12988 | 12990 | 12992 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| 12995 | 12997 | 12998 | 12999 | 13350 | 13383 | 13384 | 13425 | 13584 | 13599 | 13663 | 13687 | 13742 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| 13799 | 13845 | 13888 | 13895 | 13950 | 13991 | 13992 | 13995 | 13997 | 13999 | 14000 | 14275 | 14299 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 3 | 1 | 4 | 4 | 1 | 1 | 1 | 1 | 1 |

| 14355 | 14477 | 14480 | 14495 | 14499 | 14549 | 14677 | 14699 | 14761 | 14893 | 14900 | 14906 | 14989 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

| 14990 | 14992 | 14995 | 14999 | 15298 | 15499 | 15500 | 15688 | 15889 | 15899 | 15980 | 15988 | 15992 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 1 | 2 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 2 |

| 15995 | 15999 | 16000 | 16950 | 16992 | 16995 | 17000 | 17495 | 17500 | 17809 | 19995 | 20995 | 21992 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# Finding the mode *(cont.)*

The dataset usedcars includes three categorical variables: model, color, and transmission. Because we used the stringsAsFactors = FALSE while loading the dataset, R has left the categorical variables as character type vectors rather than converting them into factor type.

Summary statistic do not work for categorical data. Instead tables can be used to examine categorical variables. A table that presents a single categorical variable is called ***one-way table***. *The table()* function can be used to generate one-way tables. For example:

```
> table(usedcars$year)
2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
   3    1    1    1    3    2    6   11   14   42   49   16    1
```

The one way table has two rows. The first row include all different values of the categorical variable and the second row includes the count of each value. By looking at the table, you can conclude that the mode for the year is 2010.

The next command returns the proportion of each count:

```
> table_year <- table(usedcars$year)
> prop.table(table_year)
```

# Categorical variables

```
 2000       2001       2002       2003       2004       2005       2006
 2.0000000  0.6666667  0.6666667  0.6666667  2.0000000  1.3333333  4.0000000
 2007       2008       2009       2010       2011       2012
 7.3333333  9.3333333 28.0000000 32.6666667 10.6666667  0.6666667
```

Using proportions, one can conclude that 32% of the cars were made in 2010.

The following command shows the percentages directly:

```
> percent_table_year <- prop.table(table_year)*100
> percent_table_year
2000       2001       2002       2003       2004       2005       2006
 2.0000000  0.6666667  0.6666667  0.6666667  2.0000000  1.3333333  4.0000000
2007       2008       2009       2010       2011       2012
 7.3333333  9.3333333 28.0000000 32.6666667 10.6666667  0.6666667
```

One can limit the length of the fractional part which is displayed by specifying how many digits must be displayed:

```
> percent_table_year <- round(percent_table_year, digits=1)
> percent_table_year
2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012
 2.0  0.7  0.7  0.7  2.0  1.3  4.0  7.3  9.3 28.0 32.7 10.7  0.7
```

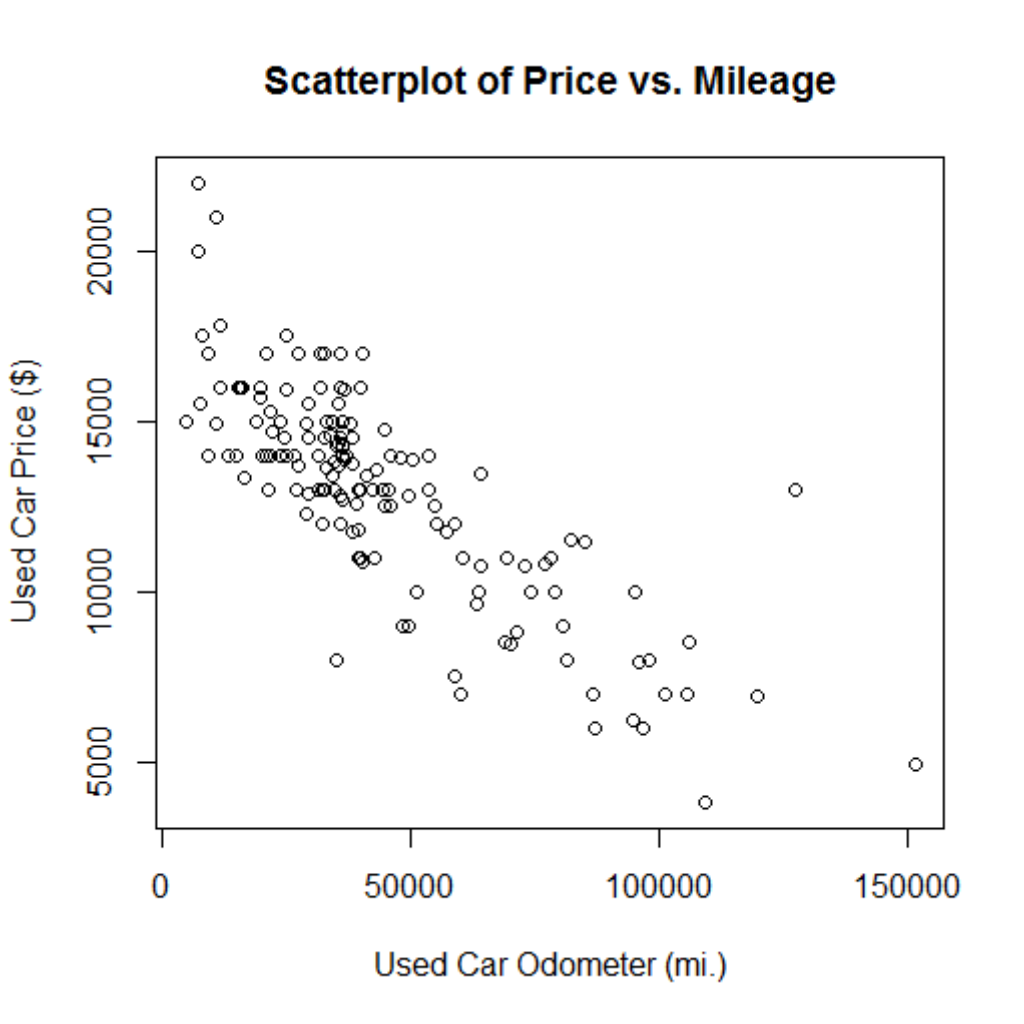
# Scatterplots

Bivariate relationships (relations between two variables) can be studied fro possible correlation using scatterplots, two dimensional diagrams in which dots are drawn on a coordinate plane using the values of one feature to provide the horizontal *x* coordinates and the values of another feature to provide the vertical *y* coordinates.

For example, if we want to find if there is a relationship between price and mileage, we can draw a scatterplot having mileage on the x axis and price on the y axis:

```
> plot(x = usedcars$mileage, y = usedcars$price, # defines each axis
+ main = "Scatterplot of Price vs. Mileage",  # the title of the scatterplot
+ xlab = "Used Car Odometer (mi.)",       # the label for the x axis
+ ylab = "Used Car Price ($)")            # the label for the y axis
```

# Scatterplots *(cont.)*



# Scatterplots *(cont.)*

The scatterplot shows a negative correlation between price and mileage: low mileage goes with high price. In other words, the dots are arranged along a diagonal running from the upper left corner to the lower right corner.

The scatterplot also shows several outliers with relatively high mileage and high price or with low mileage and low price. For example, record #90:

```
> usedcars[90,]
   year model price mileage color transmission
90 2008  SE   12995  127327   Red        AUTO
```

shows a car with relatively high mileage of 127327 and relatively high price of $12995.

# Crosstabs

***Crosstabs*** (or contingency tables) are used to study the relationship between two nominal variables. A crosstab  is a table in which the rows are the levels of one variable, while the columns are the levels of another. Counts in each of the table's cells indicate the number of values falling into the particular row and column combination.

The best way to build a crosstab is to use the *CrossTable()* function from the package *gmodels*. The function produces crosstabulations modeled after PROC FREQ in SAS or CROSSTABS in SPSS. To use the function you need to install the gmodels package:

```
> install.packages("gmodels")
```

The command will also install two dependent packages, *gtools* and *gdata*. Remember that installing a package only means to download and save the package. To load the package into working memory, you need to run:

```
> library(gmodels)
```

Let’s build a crosstab for the color and model variables. In order to limit the number of colors, we will create a new variable called conservative in usedcars. The variable conservative is logical and is TRUE id and only if the color is Black, Gray, Silver, or White.

# Crosstabs *(cont.)*

```
> usedcars$conservative <- usedcars$color %in% c("Black",
"Gray", "Silver",
+ "White")
```

The expression on the right hand side of the assignment specifies a logical condition which is evaluated for each car in usedcars. Depending on the result of the logical condition, the new variable usedcars$conservative assumes a value TRIE or FALSE.

```
> usedcars
```

| | year | model | price | mileage | color | transmission | conservative |
|---|---|---|---|---|---|---|---|
| 1 | 2011 | SEL | 21992 | 7413 | Yellow | AUTO | FALSE |
| 2 | 2011 | SEL | 20995 | 10926 | Gray | AUTO | TRUE |
| 3 | 2011 | SEL | 19995 | 7351 | Silver | AUTO | TRUE |
| 4 | 2011 | SEL | 17809 | 11613 | Gray | AUTO | TRUE |
| 5 | 2012 | SE | 17500 | 8367 | White | AUTO | TRUE |
| 6 | 2010 | SEL | 17495 | 25125 | Silver | AUTO | TRUE |
| 7 | 2011 | SEL | 17000 | 27393 | Blue | AUTO | FALSE |
| 8 | 2010 | SEL | 16995 | 21026 | Silver | AUTO | TRUE |
| 9 | 2011 | SES | 16995 | 32655 | Silver | AUTO | TRUE |

# Crosstabs *(cont.)*

```
> table(usedcars$conservative)
```

The table() function creates a table of two rows. The first row includes all levels of the conservative variable (TRUE or FALSE) and the second row includes the levels' counts:

```
FALSE  TRUE
   51    99
```

The following command creates a crosstab of models and conservative colors:

```
> CrossTable(x = usedcars$model, y = usedcars$conservative)
```

The rows in the table indicate the three models of used cars: SE, SEL, and SES. The columns indicate whether or not the car's color is conservative. The first value in each cell indicates the number of cars with that combination of model and color. The proportions indicate that the cell's proportion is relative to the chi-square statistic, row's total, column's total, and table's total. For example, 0.653 (or 65.3%) of SES cars have a conservative color.

The chi-square values refer to the cell's contribution in the Pearson's Chi-squared test for independence between the two variables. The test measures how likely it is that the difference in the cell counts in the table is due to chance alone. If the probability is very low, it provides strong evidence that the two variables are associated.

# Crosstabs *(cont.)*

```
   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  150


                 | usedcars$conservative
usedcars$model   |     FALSE |      TRUE | Row Total |
-----------------|-----------|-----------|-----------|
              SE |        27 |        51 |        78 |
                 |     0.009 |     0.004 |           |
                 |     0.346 |     0.654 |     0.520 |
                 |     0.529 |     0.515 |           |
                 |     0.180 |     0.340 |           |
-----------------|-----------|-----------|-----------|
             SEL |         7 |        16 |        23 |
                 |     0.086 |     0.044 |           |
                 |     0.304 |     0.696 |     0.153 |
                 |     0.137 |     0.162 |           |
                 |     0.047 |     0.107 |           |
-----------------|-----------|-----------|-----------|
             SES |        17 |        32 |        49 |
                 |     0.007 |     0.004 |           |
                 |     0.347 |     0.653 |     0.327 |
                 |     0.333 |     0.323 |           |
                 |     0.113 |     0.213 |           |
-----------------|-----------|-----------|-----------|
    Column Total |        51 |        99 |       150 |
                 |     0.340 |     0.660 |           |
-----------------|-----------|-----------|-----------|
```

# Crosstabs *(cont.)*

If you run the same command with an attribute chisq=TRUE:

```
>  CrossTable(usedcars$model, usedcars$conservative,
chisq=TRUE)
```

At the bottom of the crosstab, you will get:

```
Statistics for All Table Factors


Pearson's Chi-squared test
------------------------------------------------------------

Chi^2 =  0.1539564     d.f. =  2     p =  0.92591
```

The probability of 0.92591 is high. Therefore, it is likely that the variations in cell count are due to chance alone and not due to a true association between the model and the color.

Practitioners consider a "cutoff" point of 0.05 for the p-value is 0.05. That is a p value below 0.05 indicates dependence (or association) between rows and columns.

# Crosstabs *(cont.)*

A condensed form of the crosstab can be created as follows:

```
> mytable <- table(usedcars$model, usedcars$conservative)
> mytable

     FALSE TRUE
  SE     27   51
  SEL     7   16
  SES    17   32
```

Now, if you run the Chi-squared test directly on *mytable*, you will get the same result:

```
> chisq.test(mytable)

        Pearson's Chi-squared test

data:  mytable
X-squared = 0.15396, df = 2, p-value = 0.9259
```