# R Tutorial

Gerardo Ferrara*
Master in Economics and Complexity
University of Turin

## Contents



*For the most recent version of this tutorial, its corresponding data files, and related examples, please refer to the following web page: www.gerardoferrara.com

# 1 Introduction

R is an open-source computer language for statistical computing widely used in academia and increasingly in business. The R software was initially written by Ross Ihaka and Robert Gentleman in the mid 1990s. Since 1997, the R project has been organized by the R Development Core Team. It provides a coherent, flexible system for data analysis that can be extended as needed.

As a result of continuing improvements in the efficiency of R's coding and memory management, R's routines can readily process very large data sets.[1] With very large datasets, the main issue is often manipulation of data, and systems that are specifically designed for such manipulation may be preferable.

This tutorial aims to be useful as an accompaniment to a standard introductory statistics book. It is designed to allow individuals who have a basic grounding in statistical methodology to work through examples that demonstrate the use of R for a range of types of data manipulation, graphical presentation and statistical analysis.

## 1.1 Why R?

R is an extremely powerful program and includes some features that make it easy to extend or to share knowledge among users. Here are points that potential users might note:

- R has extensive and powerful graphics abilities that are tightly linked with its analytic features.
- The software is developing rapidly; new features and abilities appear every few months.
- It is just as reliable as any statistical software that is available and is exposed to higher standards of scrutiny than most other systems.
- The R community is widely drawn from application area specialists as well as statistical specialists.

## 1.2 Why not just use a spreadsheet in Excel?

Spreadsheet programs, like Excel, are tools that everyone uses, and have a friendly visual interface. Why don't we just use them? Here are three good reasons:

- Excel is not efficient when we need to analyze large datasets, with many variables (columns) and 10,000 or more observations (rows). Furthermore, it is not possible to calculate a correlation matrix for columns

[1] For example, on a Unix machine with 2GB of memory, we can efficiently compute a regression with 500,000 cases and 100 variables.

that are not contiguous. The same holds for dependent variables in a linear regression.

- Excel has limited built-in statistical tools.[2]
- Excel has limited data manipulation capabilities.

# 2 The basics

The software is obtained from the CRAN (Comprehensive R Archive Network), which may be reached from the web site http://www.r-project.org. It is sufficient to follow the installation instructions appropriate for your operating system. Packages that do not come with the base distribution must be downloaded and installed separately.

R is a functional language.[3] It is a language that uses standard forms of algebraic notation, allowing the calculations such as $1 + 2$, or $2^8$. Beyond this, most computation is handled using functions. The action of quitting from an R session uses the function call **q()**.

## 2.1 The window and menu

The window automatically opens when you start R. While a command-driven program seems an anachronism, being able to type commands and save them in file (called "Rd Document") has many advantages:

- You can save all the commands that lead to your results and also add comments to it.[4]
- Some text editors recognize R language and structure making it easy to locate errors (e.g. UltraEdit).
- You can redo the analysis with little extra work if you change data.
- You can set up an example analysis and use it as a template for other analyses.

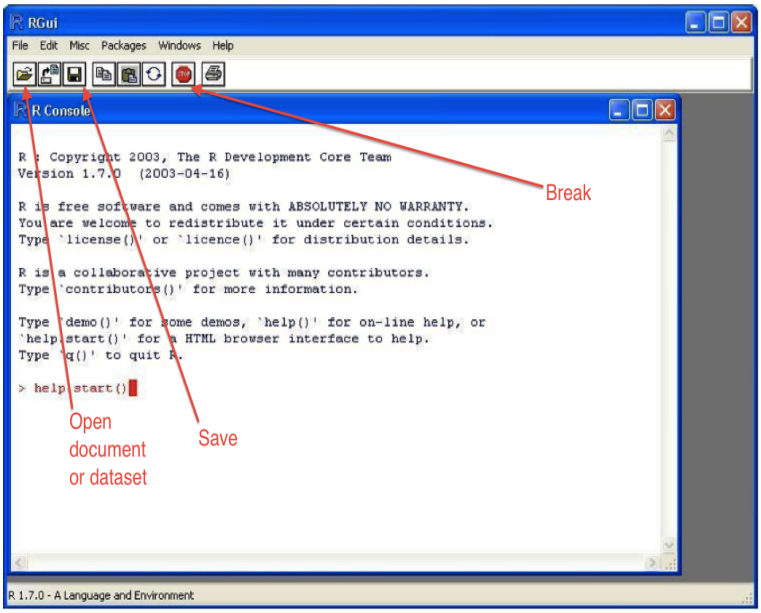
Like most programs, R has a toolbar and a menu bar with pull-down menus that you can use to access many of the features of the program. The toolbar contains buttons for more commonly used procedures. To see what each button does, hold the mouse over the button for a moment and a description of what the button does will appear. The screenshot in figure 1

[2] VBA is mandatory to expand its capabilities.

[3] The structure of an R program has similarities with programs that are written in $C^{++}$. Important differences are that R has no header files, most declarations are implicit, there are no pointers, and vectors of text strings can be defined and manipulated directly.

[4] Everything after "#" is ignored.

**Figure 1: R Toolbar**
The screenshot below highlights some of the icons available in the toolbar.




highlights some of the icons available in the toolbar. Among other things it has:

- The Open and Save icons.
- The Break icon, which allows you to interrupt (not suspend) R from processing commands.

Furthermore, Table 1 provides a summary of the main pull-down menus and their functions. The R Graphics window opens automatically when you create a graph. To bring a graph to the foreground, click on the graphics window or go to the pull-down menus and choose the R Graphics window. Graphs can be saved in various formats, such as jpg, png, bmp, ps, pdf, and so on.

When you open R, you start with a window called R Console. The console is the place where all the numerical outputs are printed. You can actually run all of your analyses and computations from the console, but once you close the software, everything that is printed in the console window (not the workspace) is lost. Therefore I strongly recommend you to run everything from a script file that you can save and load. A script is a text file with the *.Rd* extension in which you write commands meant to be sent to R. You create a script file through **File -> New script**. Notice that a script always opens in a new window, also embedded in the RGui window, which is called Untitled until you save it through **File -> Save as** and give it a name. You load a previously saved script through **File -> Open script**.

**Table 1: Summary of the main pull-down menus**
R has a menu bar with pull-down menus that you can use to access many of the features of the program.

| Menu | Functions |
|---|---|
| **File** | Open source R code; create; open and save script; load and save workspace; load and save history; display files; change working directory; print files; exit R. |
| **Edit** | Copy; paste; select all; clear console; data editor; R configuration window editor. |
| **Misc** | Stop current computation; buffer output; list objects in the memory; remove all objects; list search path. |
| **Packages & Data** | Load; install; update packages; set CRAN mirror; select repositories; install packages; install packages from local zip files. |
| **Window** | Cascade and tile R console windows; arrange icons and switch among windows. |
| **Help** | Get help on R procedures and commands; connect to the R website for more help information. |

## 2.2 Interrupt current R computation

On occasion, you could need to interrupt R computations. To stop R, either:

- Click on the red stop octagon.
- Press the ESC button on your keyboard.

## 2.3 Accessing Help in R

R provides help files for all its functions. To access a help file, use the Help pull-down menu, or type **?** followed by a function name in the R console. For example, to get help on the "print" function, type:

```
> ?print
```

There is extensive online help in the R system. The best starting point is to run the function *help.start*. This will launch a local page inside your browser with links to the R manuals, R FAQ, a search engine and other links. In the R Console the function help can be used to see the help file of a specific function (e.g. mean).

```
> help(mean)
```

We can also use the function *help.search* to list help files that contain a certain string (e.g. corr).

```
> help.search("corr")
```

# 3 Getting data into R

R generates and stores datasets in *.txt* format (with extension). Data can also be found in formatted data files, such as a file of numbers for a single data set, a table of numbers, or a file of comma-separated values. Even Excel sheets can be imported without problems. Moreover, specialized programs could convert files from one format to another (e.g. Stat/Transfer).

## 3.1 Reading data sets

There are many ways to open a data file. In addition to new functions, many packages contain a collection of built-in data sets that can be referenced by name. For example, to load a data set without uploading the entire package, could be done in this manner:

```
> data(SP500, package="MASS")
```

However, this will not load in the help files, if present, for the data set or the rest of the package. Alternately, to load a data set including its package, we could write:

```
> library(MASS)
> data(SP500)
```

See Table 2 for more details on *data* and *library*. Usually, data sets that store several variables are stored as data frames. This format combines many variables in a rectangular grid, like a spreadsheet, where each column is a different variable, and usually each row corresponds to a different time period. This conveniently allows us to have all the data vectors together in one object.

R can read many different formats of data, but as far as this course is concerned, you will only use the simplest format which is a text file with the *.txt* extension. To import data in the R environment you use the **read.table(file.choose(), header=TRUE)** command. After sending the command, a dialog window pops up and you can browse your data file. We set the optional argument header to **header=TRUE** if your data file features variable names in the first row, otherwise we write **header=FALSE** in the function. For a complete list of possible arguments to the *read.table*

command, type **?read.table** in the R console. Most arguments are optional, with the exception of the file name from which to read the data. See Table 3 for more details on *read.table*

**Table 2: library() and data()**
When dealing with data sets and packages, here is a list of other basic commands and their effect:

| Command | Effect |
|---|---|
| **library()** | List all the installed packages. |
| **library(pkg)** | Load the package *pkg*. Use **lib.loc=argument** to load package from a non-privileged directory. |
| **data()** | List all available data sets in loaded packages. |
| **data(package="pkg")** | List all data sets for this package. |
| **data(ds)** | Load the data set ds. |
| **data(ds,package=("pkg"))** | Load the data set from package. |
| **?ds** | Find help on this data set. |
| **update.packages()** | Contact CRAN and interactively update installed packages. |
| **install.packages(pkg)** | Install the package named *pkg*. This gets package from CRAN. Use **lib=argument** to specify a non-privileged directory for installation. |

**Table 3: Some commonly used arguments for the command *read.table***
When dealing with data sets, here is a list of other basic arguments and their effect:

| Argument | Effect |
|---|---|
| **file** | The complete path and filename of the file to be read. |
| **header** | Indicates whether the first line of the file is a header line, containing the names of the columns. By default, this is **FALSE**. To indicate that a file contains a header row, use **header = TRUE**. |
| **sep** | The character acting as a separator between values within a row. Often this will be a comma, a space, or a tab. To indicate a tab, use $\backslash t$; to indicate any whitespace, use $\backslash w$. |
| **nrows** | An integer value specifying how many rows should be read. By default, **read.table** will read the entire file. The **nrows** argument allows you to specify a smaller number, which will start from the first line of the file. |
| **skip** | An integer value specifying how many rows should be skipped from the top of the file. By default, **read.table** will start from the first line. The **skip** argument can be useful if a file contains multiple lines of header information that should not be read in. |
| **dec** | The character used to indicate a decimal. By default, this is a dot: "." |

### 3.2 Reading data from a line

Use *scan* to read data from a line. The following is an example that reads price and volume data of 4 periods and puts them into a dataset named "sample". This strategy makes the most sense when you have a small amount of data to read in. The variables price and volume can be created, as follow:

```
> price <- scan()
> 1: 0.9 1.2 1.1
> 4:
> volume <- scan()
> 1: 100 200 150
> 4:
> sample <- data.frame (price, volume)
```

Here, R is prompting you to enter another value, which would be the 4th value in the vector you are entering. If you are finished entering data, hit return again. After we call *scan*, R prompts you with the 1:, meaning the first thing you enter will be the first element in the vector of values.

Alternatively, vectors can be made with the **c ()** function, which combines its arguments. For example:

```
> price = c(0.9, 1.2, 1.1)
> volume = c(100, 200, 150)
```

### 3.3 Saving data sets

We usually use *write.table* to export an R dataset into a different format. The **write.table()** function has many of the same arguments as **read.table()**. For example, we can save the file *mytrial* as a ","-delimited textfile format to the C:/Mydata directory with this command:

```
> write.table(mytrial, "C:/Mydata/mytrial.txt", sep = ",")
```

To export it as .csv format, you can use the write.csv variant:

```
> write.csv(mytrial, "C:/Mydata/mytrial.csv")
```

Here, by default, **sep = ","** and **dec = "."**.

# 4 Saving your commands

By default, R keeps track of all the commands you use in a session. This tracking can come in handy if you need to reuse a command you used earlier or want to keep track of the work you did before. These previously used commands are kept in the history.

You can browse the history from the command line by pressing the up-arrow and down-arrow keys. When you press the up-arrow key, you get the commands you typed earlier at the command line. You can press Enter at any time to run the command that is currently displayed.

Saving the history is done using the *savehistory* function. By default, R saves the history in a file called .Rhistory in your current working directory. This file is automatically loaded again the next time you start R, so you have the history of your previous session available. If you want to use another filename, use the argument file as follow:

```
> savehistory(file = "another.Rhistory")
```

Alternatively, on a PC you can also access this through the menu **File -> Save history...** and browse to the folder where you want to save the file and give it a name..
Furthermore, we can take a look at the history by opening the file in a normal text editor, like Notepad. If you want to load a history file you saved earlier, you can use the *loadhistory* function. This will replace the history with the one saved in the .Rhistory file in the current working directory. If you want to load the history from a specific file, you use the file argument again, as follow:

```
> loadhistory("another.Rhistory")
```

On a PC you can also access this through the menu **File -> Load history** and browse to the folder where you saved the .Rhistory file and click open.

You can save all the objects (vectors, matrices, data frames, and lists) and functions that you have created in an .RData file, by using the *save* function:

```
save("myfile.RData")
```

Alternatively, on a PC you can also access this through the menu **File -> Save workspace file...** and browse to the folder where you want to save the file and give it a name.
If you want to load an image, you use the file argument again, as follow:

```
load("myfile.RData")
```

On a PC you can also access this through the menu **File -> Load workspace file...** and browse to the folder where you saved the .RData file and click open.

# 5 Means, correlations, and other statistics

R has many functions for statistical analyses. You can perform many statistical procedures in R by typing commands in the R console or by running the scripts in the R Editor window. Some advanced functions need to be installed from the package before they can be executed. Please also keep in mind that the R language is case sensitive.

## 5.1 Summary: What is in the data set?

To get quick descriptive statistics for all variables in the dataset, we can use the summary function:

```
> data(SP500, package=("MASS"))
> summary(SP500)
Min.      1st Qu.   Median   Mean     3rd Qu.   Max.
-7.11300  -0.41410  0.04210  0.04575  0.54280   4.98900
```

## 5.2 Descriptive statistics

We can get individual descriptive statistics (*mean, variance, standard deviation, minimum and maximum*) of a single variable:

```
> mean(SP500)
0.04575
> var(SP500)
0.8982233
> sd(SP500)
0.9477464
> min(SP500)
-7.11300
> max(SP500)
4.98900
```

The *cor* function will give the correlation matrix of all variables in the dataset. We can also get the correlation matrix of the second, third, and fourth variables in the dataset:

```
> data(waders, package=("MASS"))
> cor(waders [c(2,3,4)])
1.0000000 -0.2057986 -0.2072291
-0.2057986 1.0000000 0.6971051
-0.2072291 0.6971051 1.0000000
```

# 6 Regression Analysis

## 6.1 Simple linear regression model

We use the command $lm(Y \sim X + I)$ to perform a linear regression, where $Y$ is the dependent variable and $X$ and $I$ are both independent variables. The $\sim$ (tilde) is read "is modeled by" and is used to separate the response variable from the predictor. We can perform a linear regression example, as follow:

```
> data(mammals, package=("MASS"))
> rel <- lm(brain ~ body, data=mammals)
> summary(rel)
```

| | Estimate | Std. Error | t value | Pr(>\|t\|) |
|---|---|---|---|---|
| (Intercept) | 91.0044 | 43.5526 | 2.09 | 0.0409 |
| body | 0.9665 | 0.0477 | 20.28 | 0.0000 |

Residual standard error: 334.7 on 60 degrees of freedom
Multiple R-squared: 0.8727, Adjusted R-squared: 0.8705
F-statistic: 411.2 on 1 and 60 DF, p-value: < 2.2e-16

The *summary* function for the linear regression model displays the regression model with the residuals, estimated coefficients, and R-squared value. By default, the *lm* function will print out the estimates of the coefficients. We can also get more information as needed by using extractor functions listed in Table 4.

**Table 4: Some commonly used extractor functions**
When dealing with predictive models, here is a list of extractor functions and their effect:

| Funtion | Effect |
|---|---|
| **summary** | It is a generic function used to produce result summaries of the results of various model fitting functions. |
| **plot** | Generic function for plotting R objects. |
| **coef** | It is a generic function which extracts model coefficients from objects returned by modeling functions. |
| **resid** | It is a generic function which extracts model residuals from objects returned by modeling functions. All object classes which are returned by model fitting functions should provide a residuals method. |
| **fitted** | It is a generic function which extracts fitted values from objects returned by modeling functions. |
| **deviance** | Returns the deviance of a fitted model object. |
| **predict** | It is a generic function for predictions from the results of various model fitting functions. Most prediction methods which are similar to those for linear models have an argument newdata specifying the first place to look for explanatory variables to be used for prediction. Time series prediction methods in package stats have an argument n.ahead specifying how many time steps ahead to predict. |
| **anova** | Compute analysis of variance tables for one or more fitted model objects. When given a single argument it produces a table which tests whether the model terms are significant. When given a sequence of objects, anova tests the models against one another in the order specified. |

We can also use the *plot* function to get diagnostic informations for *lm* objects:

```
> plot(rel)
```

where *rel* represents the regression above. Figure 2 shows the basic diagnostic graphs for our regression model: residual versus fitted plot, normal qq-plots of the residuals, standardized residual versus fitted plot, and scatter plots with the regression line overlaid.

Linear regression is the name of a procedure that fits a straight line to the data. The line is used to predict the value of $y$ (brain weight) for a known value of $x$ (body weight).[5] Suppose we write the equation of the line as:

$$y = a + bx \tag{1}$$

The regression is used to choose the values of $a$ and $b$ that minimize the sum or the squares of the residual errors. Using the *abline* command we get the graphical representation of our regression:

[5]The variable $x$ is the predictor variable and $y$ the response variable.

**Figure 2: Diagnostic plot**
The plot method gives the diagnostic information for *lm* objects. By default the first, second and fourth plot use the row names to identify the three most extreme residuals.

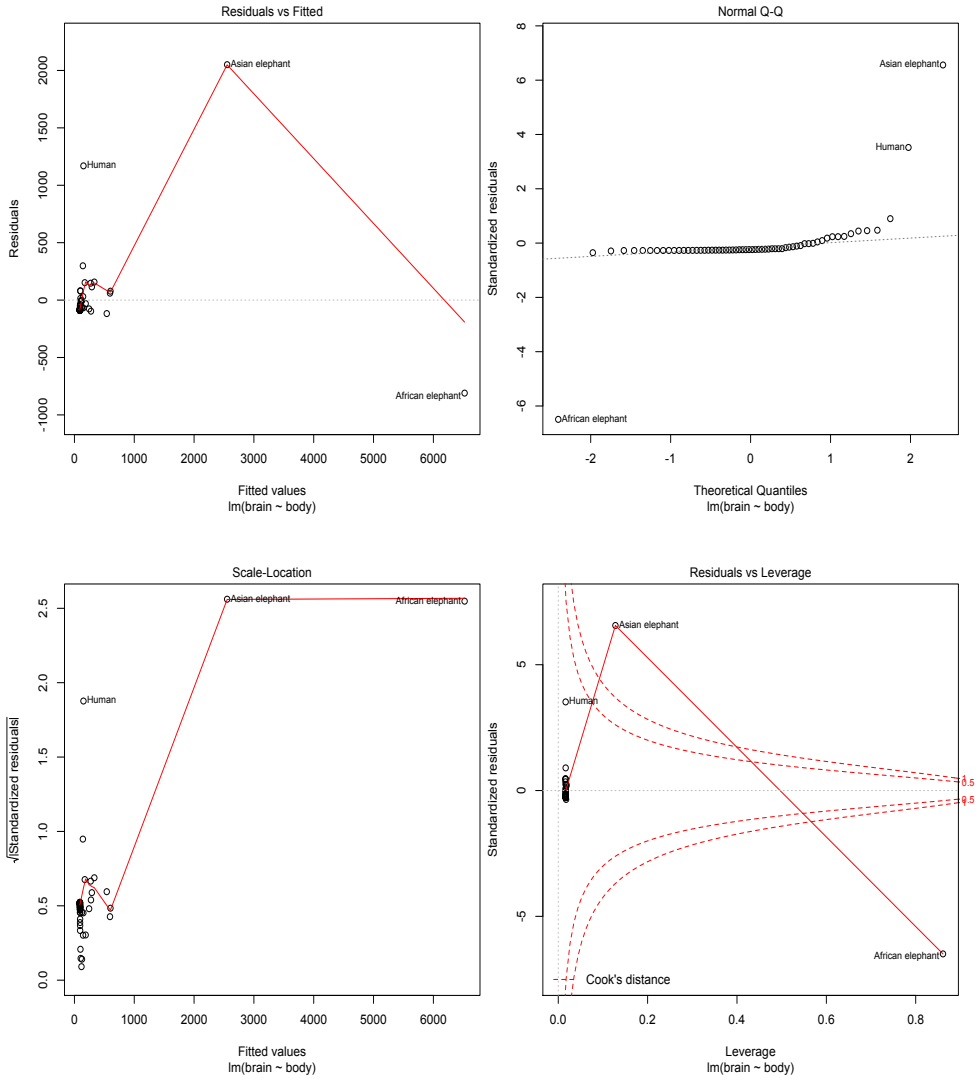
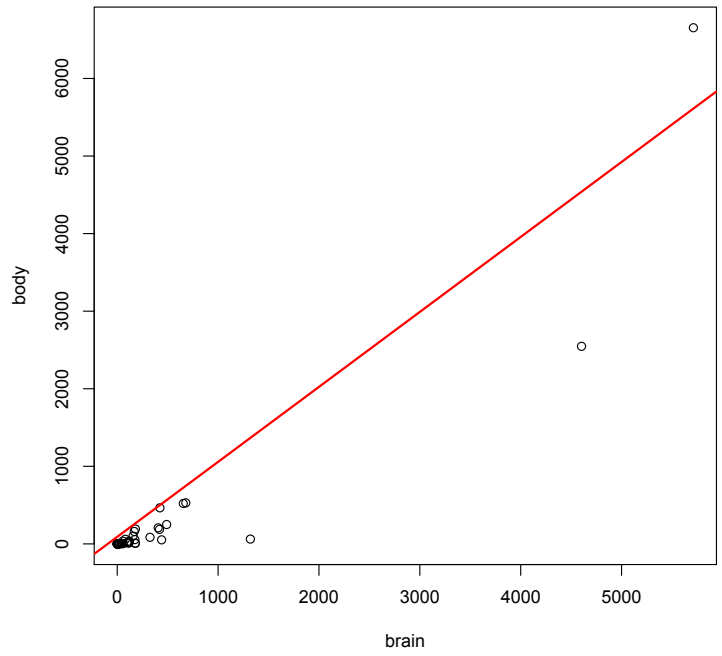


**Figure 3: Mammals data with regression line**
As suggested by the scatterplot, there is a linear (and positive) relation.




```
> plot(brain, body); abline(rel, col=2, lwd=2)
```

where the *plot* command creates a graph window with a scatterplot. Figure 3 offers a representation of our regression goodness.

Given this model, one can predict the brain weight based on our data set. We can simply use the *predict* command.[6] For example, the predicted brain weight would be:

```
> rel <- lm(brain ~ body, data=mammals[1:50,1:2]);
> pred<- predict(rel, mammals[51:62,1:2]);
```

where to explore the prediction problem, we hold just a part of this data set and use the remaining data to make the prediction.

[6]We need a data frame with column names matching the predictor or explanatory variable.

# 7 Charts

Summary statistics from our data set can be represented as charts using R's graph features. In this section we will first discuss the graphical functions that can be found in the base R system and the *lattice* package. *Lattice* is an excellent package for visualizing multivariate data, which is essentially a great set of routines for quickly displaying complex data. There are more R graphics facilities that can be used in both interactive and batch modes, but in most cases, interactive use is more productive. This makes it ideal for use in exploratory data analysis.

## 7.1 Scatterplots

There are many ways to create a scatterplot in R. The basic function is **plot**$(\mathbf{x}, \mathbf{y})$, where $x$ and $y$ are numeric vectors denoting the $(x, y)$ points to plot. Furthermore, if we need to identify a particular point in a plot, the *identify* function can be used as follow:

```
> attach(mtcars);
> plot(wt, mpg, main="Example", xlab="Weight ", ylab="Miles");
> with(identify(wt, mpg,n=2, labels=rownames(mtcars)))
```

now left click near two points and the element name of that point will be printed at the bottom, left, top or right of the point, depending on which side of it you clicked. Figure 4 shows the basic scatterplot for our data frame.

## 7.2 Histograms and density plots

We can create histograms with the *hist* function . The option **freq=FALSE** plots probability densities instead of frequencies. The option **breaks=...** controls the number of bins. Figure 5 with a basic example of histogram is created as follow:

```
> hist(mpg)
```

We can also create a colored histogram with different number of bins, as follow:

```
> hist(mtcars$mpg, breaks=12, col="red")
```

Figure 6 displays the colored histogram for the example above. Histograms

**Figure 4: Enhanced Scatterplot**
A scatterplot provides a graphical view of the relationship between two sets of numbers. Here we provide examples using the *mtcars* data frame which is mentioned above. In particular we look at the relationship between the"Weight of Car" and the "Miles Per Gallon".

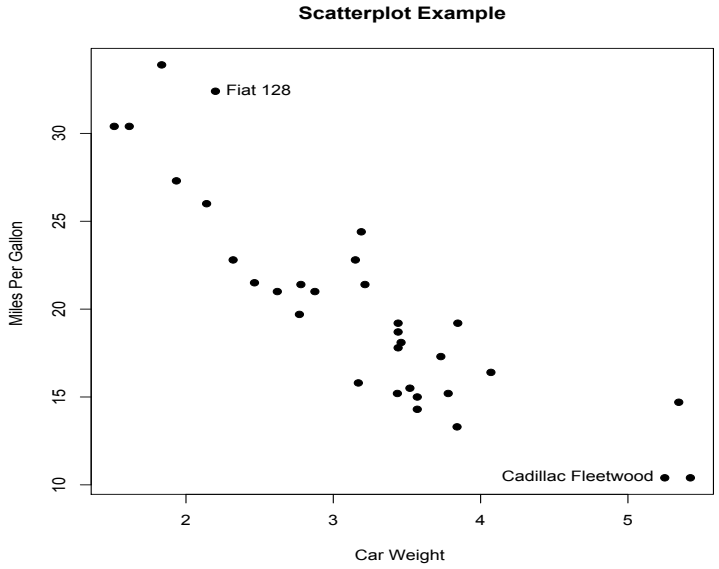



**Figure 5: Basic histogram**
An histogram provides a graphical view of the given data values. R's default with equi-spaced breaks is to plot the counts in the cells defined by breaks. Thus the height of a rectangle is proportional to the number of points falling into the cell, as is the area provided the breaks are equally-spaced. Here we provide examples using the *mtcars* data frame which is mentioned above.

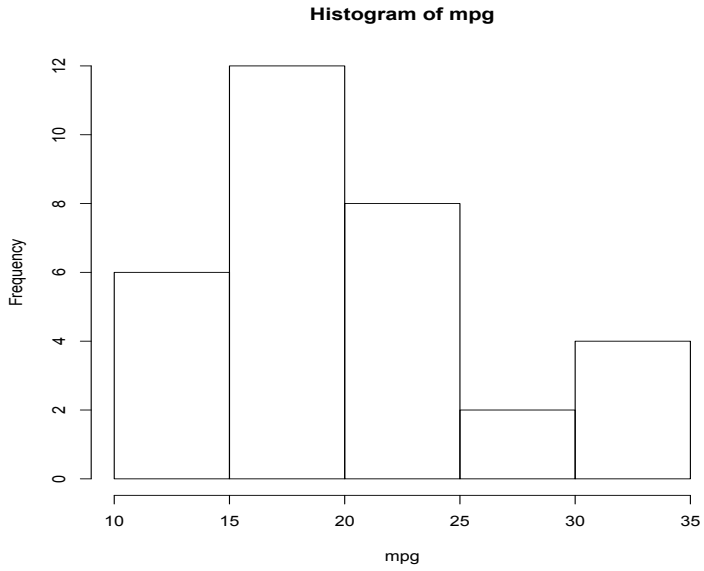
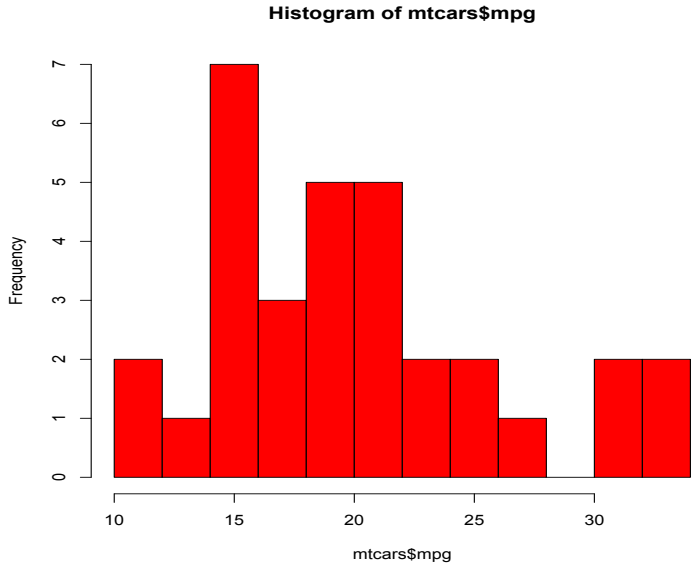


**Figure 6: Enhanced histogram**
A colored histogram is a representation of tabulated frequencies, shown as adjacent rectangles, erected over discrete intervals (bins), with a colored area equal to the frequency of the observations in the interval.




are used to plot the density of data, and often for density estimation. However, it can be a poor method for determining the shape of a distribution because it is so strongly affected by the number of bins used. The only difference between Figure 5 and Figure 6 is that a different choice of breakpoints is used for the histogram, so that the histogram gives a rather different impression of the distribution of the data.

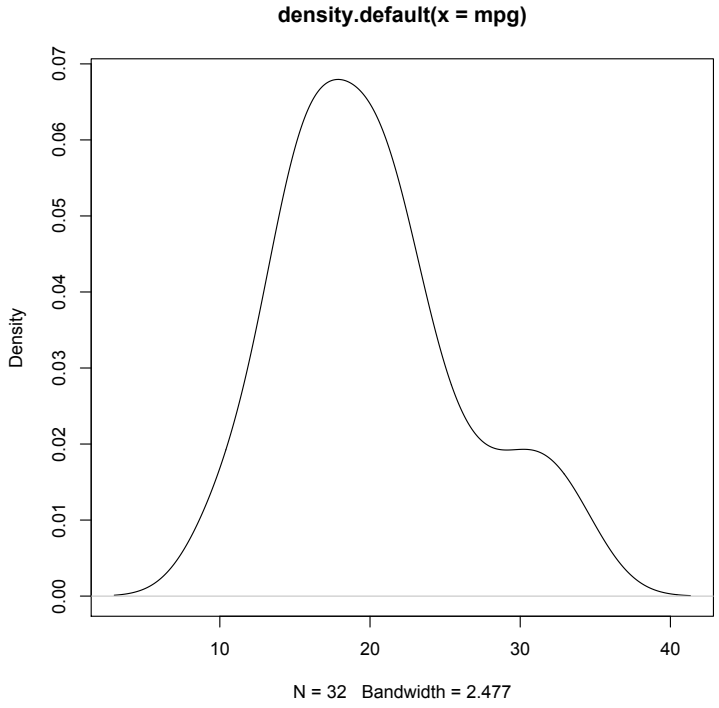
## 7.3 Kernel density plots

Kernal density plots are usually a much more effective way to view the distribution of a variable. Kernel density estimation is a non-parametric method of estimating the probability density function of a continuous random variable. It is non-parametric because it does not assume any underlying distribution for the variable. Here is the code used to plot the histogram in Figure 7:

```
> d <- density(mpg)
> plot(d)
```

where the *density* function returns the density data. As researchers, we would like to summarize data without sacrificing useful information. As seen in the previous example, a histogram can be visually frustrating and

**Figure 7: Kernel Density Plot**
We can easily compare the histogram (figure 5, figure 6) and kernel density estimate constructed using the same data. Kernal density plots are a more effective way to illustrate the distribution of a variable. The right-skewed shapes of the curve also suggest that the normal distribution may not be suitable.




misleading, especially when bins or midpoints are not appropriately sized or placed.

## 7.4 Pie charts

A pie chart of a qualitative data sample consists of pie wedges that shows the frequency distribution graphically. Figure 8 is created with the *pie* function as follow:

```
> slices <- c(10, 8,4, 20, 8)
> lbls <- c("Italy", "UK", "United States", "Germany", "France")
> pie(slices, labels = lbls, main="Pie Chart of Countries")
```

where **labels**= notes a character vector of names for the slices.

Figure 9 with annotated percentages is created as follow:

**Figure 8: Pie chart**
A pie chart for the example data set. I recommend histograms over pie charts because people are able to judge length more accurately than volume.

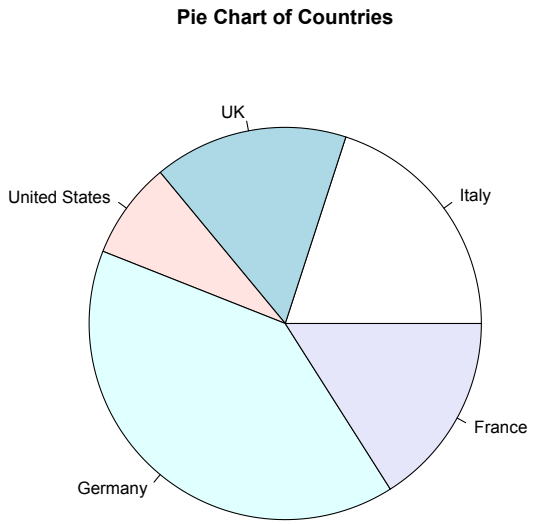
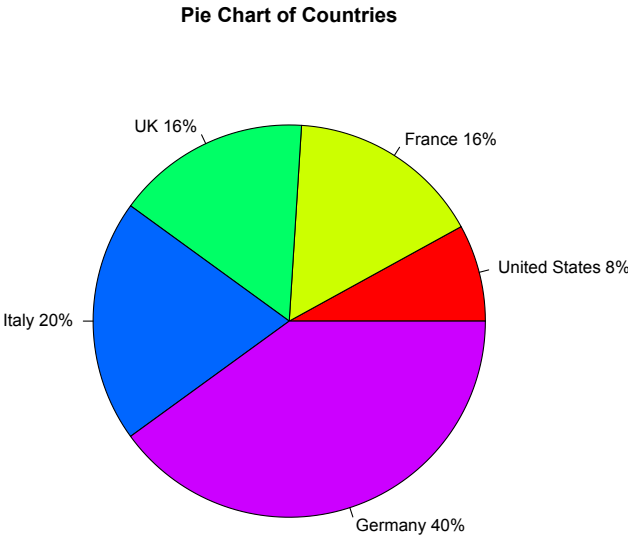



**Figure 9: Pie chart with annotated percentages**
Percentage shows each slice as a calculated percentage of the total. When drawing a pie chart, ensure that the segments are ordered by size (largest to smallest) and in a clockwise direction.



```
> slices <- c(4, 8, 8, 10, 20)
> lbls <- c("United States", "France", "UK", "Italy", "Germany")
> pct <- round(slices/sum(slices)*100)
> lbls <- paste(lbls, pct)
> lbls <- paste(lbls,"%",sep="")
> pie(slices,labels = lbls, col=rainbow(length(lbls)),
+ main="Pie Chart of Countries")
```

Pie charts are not recommended in the R documentation, and their features are somewhat limited. Pie charts are very widely used in the business world and the mass media. However, they have been criticized, and many experts recommend avoiding them, pointing out that research has shown it is difficult to compare different sections of a given pie chart, or to compare data across different pie charts.

# 8 Manipulating data sets

The programming language in R provides many different functions and mechanisms to manipulate data. Getting comfortable with viewing and manipulating multivariate data forces you to be organized about your data.

## 8.1 Extracting data from data frames

Data frames are arrays as they have columns which are the variables and rows which are for the experimental observation. Thus we can access the data by specifying a row and a column. Let's import the information in the *infert*[7] data set, an R library, so that different aspects of data frame manipulation can be demonstrated, and use the function *names* to see the column names of the *infert* data frame:

```
> attach(infert);
> names(infert)
[1] "education"      "age"          "parity"       "induced"      "case"
[6] "spontaneous"    "stratum"      "pooled.stratum"
```

To select a specific column from a data frame use the $ symbol or double square brackets and quotes:

[7] Infertility after spontaneous and induced abortion.

```
> age <- infert$age
> age <- infert[["age"]]
```

In this way, the object *age* is a vector. If you want the result to be a data frame use single square brackets:

```
> age <- infert["age"]
```

When using single brackets it is possible to select more than one column from a data frame. The result is again a data frame:

```
> age <- infert["education" , "age"]
```

Rows from a data frame can also be selected using row numbers. Selecting cases 10 trough 13 from the *infert* data frame, we get:

```
> infert[10:13,]
```

| | educ | age | par | induced | case | spont | strat | pooled |
|---|---|---|---|---|---|---|---|---|
| 10 | 6-11yrs | 28.00 | 2.00 | 0.00 | 1.00 | 0.00 | 10 | 19.00 |
| 11 | 6-11yrs | 29.00 | 2.00 | 1.00 | 1.00 | 0.00 | 11 | 20.00 |
| 12 | 6-11yrs | 37.00 | 4.00 | 2.00 | 1.00 | 1.00 | 12 | 37.00 |
| 13 | 6-11yrs | 31.00 | 1.00 | 1.00 | 1.00 | 0.00 | 13 | 9.00 |

To display specific cases from a data frame you can also use a logical vector. When you provide a logical vector in a data frame selection, only the cases which correspond with a *TRUE* are selected. Suppose you want to get all people from the *infert* data frame that have an age of over 43:

```
> trial <- infert$age > 43
> infert[trial, ]
```

| | educ | age | par | induced | case | spont | strat | pooled |
|---|---|---|---|---|---|---|---|---|
| 20 | 6-11yrs | 44.00 | 1.00 | 0.00 | 1.00 | 1.00 | 20 | 17.00 |
| 103 | 6-11yrs | 44.00 | 1.00 | 0.00 | 0.00 | 0.00 | 20 | 17.00 |
| 185 | 6-11yrs | 44.00 | 1.00 | 1.00 | 0.00 | 0.00 | 20 | 17.00 |

A handy alternative is given by the function *subset*. This allows the use of the standard indexing conventions so that for example ranges of columns can be specified easily, or single columns can be dropped:

```
> subset(infert, age>40 & spontaneous<1)
```

| | educ | age | par | induced | case | spont | strat | pooled |
|---|---|---|---|---|---|---|---|---|
| 2 | 0-5yrs | 42.00 | 1.00 | 1.00 | 1.00 | 0.00 | 2 | 1.00 |
| 33 | 6-11yrs | 42.00 | 1.00 | 1.00 | 1.00 | 0.00 | 33 | 16.00 |
| 43 | 6-11yrs | 41.00 | 1.00 | 0.00 | 1.00 | 0.00 | 43 | 15.00 |
| 85 | 0-5yrs | 42.00 | 1.00 | 0.00 | 0.00 | 0.00 | 2 | 1.00 |
| 103 | 6-11yrs | 44.00 | 1.00 | 0.00 | 0.00 | 0.00 | 20 | 17.00 |
| 116 | 6-11yrs | 42.00 | 1.00 | 1.00 | 0.00 | 0.00 | 33 | 16.00 |
| 126 | 6-11yrs | 41.00 | 1.00 | 0.00 | 0.00 | 0.00 | 43 | 15.00 |
| 167 | 0-5yrs | 42.00 | 1.00 | 0.00 | 0.00 | 0.00 | 2 | 1.00 |
| 185 | 6-11yrs | 44.00 | 1.00 | 1.00 | 0.00 | 0.00 | 20 | 17.00 |
| 198 | 6-11yrs | 42.00 | 1.00 | 0.00 | 0.00 | 0.00 | 33 | 16.00 |
| 208 | 6-11yrs | 41.00 | 1.00 | 0.00 | 0.00 | 0.00 | 43 | 15.00 |

## 8.2 Adding columns (variables) to a data frame

The function *cbind* can be used to add additional columns to a data frame. For example, the vector *children* with the number of children for each woman can be added to the *infert* data frame as follows:

```
> children=trunc(runif(248, min=0, max=9))
> new.infert <- cbind(infert, children=as.data.frame(children))
```

The left hand side of the = specifies the column name in the *infert* data frame and the right hand side is the vector you want to add. The command *trunc* takes a single numeric argument $x$ and returns a numeric vector containing the integers formed by truncating the values in $x$. Hence, we need to use *as.data.frame* function to switch an object into a data frame.

## 8.3 Combining data frames

Use the function *rbind* to combine two or more data frames. Consider the following two data frames *rand.df1* and *rand.df2*:

```
> part1 <- data.frame(norm = rnorm(5), binom = rbinom(5,7,0.8),
+ unif=runif(5))
> part1
```

| | norm | binom | unif |
|---|---|---|---|
| 1 | 0.44 | 6.00 | 0.48 |
| 2 | 0.25 | 7.00 | 0.13 |
| 3 | 1.12 | 6.00 | 0.93 |
| 4 | 1.51 | 6.00 | 0.67 |
| 5 | -0.05 | 6.00 | 0.04 |

```
> part2 <- data.frame(chisq = rchisq(5,2), binom = rbinom(5,5,0.1),
+ unif=runif(5))
> part2
```

| | chisq | binom | unif |
|---|---|---|---|
| 1 | 1.81 | 1.00 | 0.11 |
| 2 | 0.98 | 1.00 | 0.89 |
| 3 | 0.26 | 1.00 | 0.31 |
| 4 | 2.72 | 1.00 | 0.58 |
| 5 | 0.49 | 1.00 | 0.56 |

These two data frames have two columns in common: *binom* and *unif*. When we only need to combine the common columns of these data frames, you can use the subscripting mechanism and the function *rbind*:

```
> comb <- rbind(part1[ , c("unif","binom")], part2[ , + c("unif", "bi-
nom")])
> comb
```

| | unif | binom |
|---|---|---|
| 1 | 0.48 | 6.00 |
| 2 | 0.13 | 7.00 |
| 3 | 0.93 | 6.00 |
| 4 | 0.67 | 6.00 |
| 5 | 0.04 | 6.00 |
| 6 | 0.11 | 1.00 |
| 7 | 0.89 | 1.00 |
| 8 | 0.31 | 1.00 |
| 9 | 0.58 | 1.00 |
| 10 | 0.56 | 1.00 |

The function *rbind* expects that the two data frames have the same column names. The function *rbind.fill* in the *reshape* package can stack two or more data frames with any column names:[8]

```
> install.packages('reshape')
> library(reshape)
> rbind.fill(part1,part2,part1)
```

## 8.4 Merging data frames

Two data frames can be merged into one data frame using the function merge. If the original data frames contain identical columns, these columns only appear once in the merged data frame. In most cases, you join two data frames by one or more common key variables. By default the data frames are merged on the columns with names they both have, but separate specifications of the columns can be given. Let's consider the following two data frames:

```
> example1 = data.frame(CustomerId=c(1:6),Product=c(rep("Car",3),
+ rep("Radio",3)))
> example2 = data.frame(CustomerId=c(2,4,6),City=c(rep("Turin",2),
+ rep("Milan",1)))
> example1
```

| | CustomerId | Product |
|---|---|---|
| 1 | 1 | Car |
| 2 | 2 | Car |
| 3 | 3 | Car |
| 4 | 4 | Radio |
| 5 | 5 | Radio |
| 6 | 6 | Radio |

```
> example2
```

| | CustomerId | City |
|---|---|---|
| 1 | 2.00 | Turin |
| 2 | 4.00 | Turin |
| 3 | 6.00 | Milan |

[8] It will fill a missing column with NA.

To perform a complete merge of cold and large states, use *merge* function as follow:

```
> merge(example1, example2)
```

| | CustomerId | Product | City |
|---|---|---|---|
| 1 | 2 | Car | Turin |
| 2 | 4 | Radio | Turin |
| 3 | 6 | Radio | Milan |

This command will work for these examples because R automatically joins the frames by common variable names, but you would most likely want to specify the common index to make sure that you were matching on only the fields you desired.

```
> merge(example1, example2, by="CustomerId")
```

I think it's almost always best to explicitly state the identifiers on which you want to merge; it's safer if the input *data.frames* changes unexpectedly and easier to read later on.

The *merge* function takes quite a large number of arguments. There are four ways of combining data:

- Natural join: To keep only rows that match from the data frames, specify the argument **all=FALSE**.
- Full outer join: To keep all rows from both data frames, specify the argument **all=TRUE**.
- Left outer join: To include all the rows of your data frame x and only those from y that match, specify the argument **all.x=TRUE**.
- Right outer join: To include all the rows of your data frame y and only those from x that match, specify the argument **all.y=TRUE**.

## References


[1] Knell, J. R., and Braun, J., *Introductory R: A Beginner's Guide to Data Visualisation and Analysis using R*, Springer, 2013.

[2] Maindonald, J., and Braun, J., *Data Analysis And Graphics Using R*, Cambridge University Press, 2007.

[3] Verzani, J., *Using R for introductory statistics*, Chapman & Hall/CRC, Boca Raton, FL, 2005.