R Reference Card

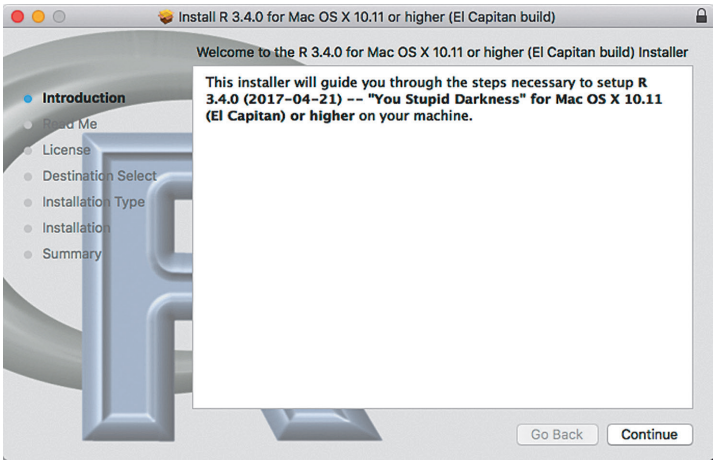
Pearson

Introduction

The programming language R is a free, open-source statistical software package. Data values can be stored in objects in R; commands run on these objects can produce graphical displays, statistical summaries, or more complex analyses. We will discuss some of the main object types and demonstrate the use of frequently used functions in R.

Installation

R can be downloaded for free at: http://cran.r-project.org/. You should choose the appropriate installation file based on your operating system. Run the installation file and complete all steps of installation.

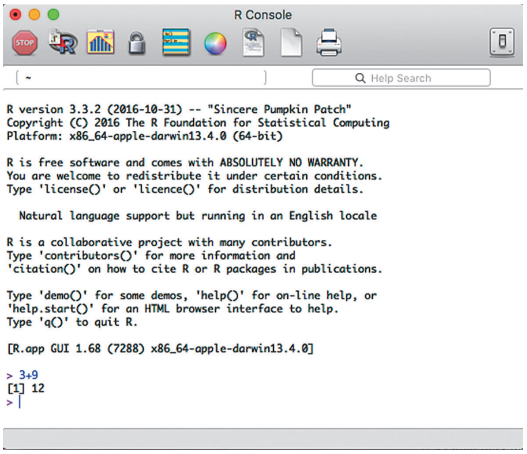


Getting Started

Once installation completes, you can open the R application. An R Console will pop up. Commands can be entered in the console line by line, as with a calculator. Lines beginning with '>' are prompts for user input; others contain R output. The simplest commands that can be run are mathematical operations. Here is a table of the frequently used mathematical operators/functions for calculations in R.

Note: R is *case-sensitive* [e.g., `exp(3)` is not equivalent to `Exp(3)`].

Operator/Function	Meaning	Example
`+, -, *, /`	The basic 4 mathematical operations: addition, subtraction, multiplication, and division. Order of operations is obeyed.	`(3+4)*9`
`^`	Exponentiation	`3^5`
`exp()`	Exponential function	`exp(3)`
`log()`	Natural log function	`log(10)`
`log10()`	Log base 10 function	`log10(10)`



ISBN-13: 978-0-13-444214-3
ISBN-10: 0-13-444214-8
EAN
9 780134 442143
90000>

Working with Data Objects

The '<-' operator is used for creating objects. The command 'x <- 3' stores the value 3 into an object named 'x'. The '=' operator can also be used (as in 'x=3'), but is generally not considered best practice.

```
> x <- 3
> x
[1] 3
>
> x <- 3+9
> x
[1] 12
```

The simplest kind of data object in R is a **vector**. 'x' is a vector of length 1. We can store the results of any earlier calculations into 'x' just as easily. Note that if we type only the object name and hit return, R will display the value stored in the object.

```
> x <- c(1, 2, 5)
> x
[1] 1 2 5
>
> y <- seq(5, 9, 1)
> y
[1] 5 6 7 8 9
```

Other ways of creating a vector include concatenating together elements via `c()` and creating a numeric vector from a sequence via `seq()`, which creates a numeric sequence spanning the range of the first two arguments and incrementing by the third argument. A shorthand for `seq(a, b, 1)` is `a:b`.

We can subset a vector (1) by referencing the indices to extract or (2) by supplying a condition within a set of square brackets [].

(The # sign denotes a comment; R ignores all text on the line following a # sign.)

```
> y <- 5:9
> y[3]          # extracts the 3rd element
[1] 7
> y[c(2, 4)]    # extracts 2nd, 4th elements
[1] 6 8
> y[y > 7]      # extracts elements > 7
[1] 8 9
```

A vector can be used to store the observed values of a single variable. Typically, datasets used in statistical analyses involve multiple variables. **Data frames** are the perfect data structure for housing such datasets. R has some built-in data frames for demonstration purposes. For example, here we have the `iris` data frame. We can run the `head()` function on any data object to display the first 6 entries. In this display, we see that there are 5 variables, stored as 5 columns, in our `iris` data frame. A data frame is a 2-dimensional object; we can run `dim()`, `nrow()`, or `ncol()` to see its dimensions.

We can extract parts of a data frame in several ways.

```
iris$Species   # extracts the column by name
iris[,5]       # extracts the column by index
iris[3,]       # extracts a row by index
iris[3,5]      # extracts a value by row
               # and column index
iris[,-5]      # extracts everything but the
               # 5th column
iris[-1,]      # extracts everything but the
               # 1st row
```

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
>
>
> dim(iris)
[1] 150   5
>
> nrow(iris)
[1] 150
>
> ncol(iris)
[1] 5
```

Importing Data

It is easy to import a .csv file as a data frame in R using the following command (column headers in the first row):

```
z <- read.csv("path_to_file/filename.csv")
```

Instead of specifying the file path as part of the function call, it is also possible to set the working directory beforehand via setwd("path_to_files").

Descriptive Statistics

The distribution of one or two categorical variables can be displayed in a frequency table using the `table()` command. If `x` and `y` are vectors of categorical variables of the same length, we can run:

```
> table(x)      # a one-way table
> table(x, y)   # a two-way table
```

For a quantitative variable `x`, frequently used summary statistics are:

5-Number Summary (and mean): `> summary(x)`
Quantiles (Percentiles): `> quantile(x, prob=a)`
where a is the desired quantile (in decimal form).

Mean: `> mean(x)`
Standard Deviation: `> sd(x)` Variance: `> var(x)`

The dependence between two quantitative variables `x` and `y` can be measured using:
Correlation: `> cor(x, y)` Covariance: `> cov(x, y)`

```
> table(iris$Species)

    setosa versicolor  virginica
        50         50         50
> summary(iris$Sepal.Length)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.300   5.100   5.800   5.843   6.400   7.900
>
> cor(iris$Sepal.Length, iris$Sepal.Width)
[1] -0.1175698
>
> quantile(iris$Sepal.Width, c(0.025, 0.975))
  2.5%  97.5%
2.2725 3.9275
```

Plotting

HISTOGRAMS `hist(x)` operates on a numeric vector x to produce a histogram. R will automatically decide how many bins to use. If you would like to suggest how many bins to use, you can use the command:

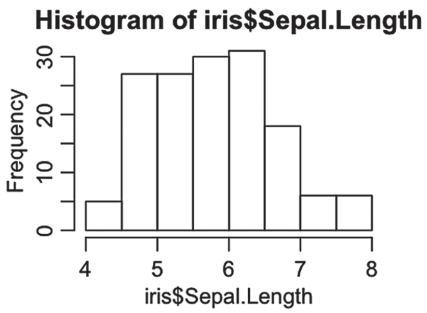
```
> hist(x, breaks = b)
```

where `b` is the number of breaks to be used. The number of bins is equal to the number of breaks plus one. Note that R will take `b` as a suggestion and may add more bins if deemed necessary.

Additionally, we can display density on the y-axis as opposed to frequency by including the argument `prob = TRUE`:

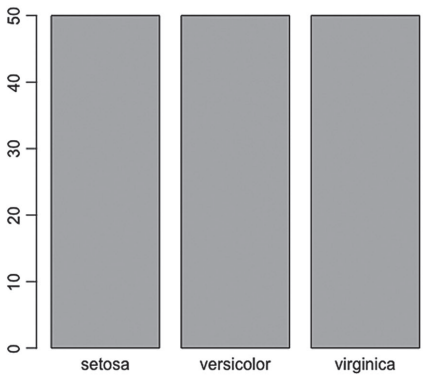
```
> hist(x, prob = TRUE)
```

```
> hist(iris$Sepal.Length)
```



BAR CHARTS The `barplot(counts)` function creates a bar chart based on a vector of `counts` of a categorical variable. An easy way to obtain these bar heights is to use the `table()` function to obtain a frequency table.

```
> barplot(table(iris$Species))
```

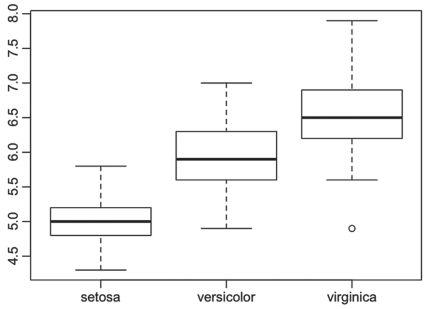


BOXPLOTS Boxplots are useful for visualizing key quantiles of a quantitative variable `y` by groupings of a categorical variable `x`. A boxplot can be displayed using the syntax `boxplot(y ~ x)`.

```
> boxplot(iris$Sepal.Length ~ iris$Species)
```

When `y` and `x` are columns in the same data frame, say, `z`, as is the case for our example above, an equivalent command is `boxplot(y ~ x, data=z)`.

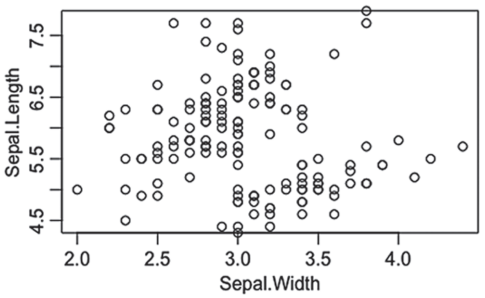
```
> boxplot(Sepal.Length ~ Species, data=iris)
```



SCATTERPLOTS Scatterplots are helpful for examining the relationship between 2 quantitative variables `x` and `y`. We use the `plot()` function, using similar syntax as the `boxplot()` function.

```
> plot(iris$Sepal.Length ~ iris$Sepal.Width)
> plot(Sepal.Length ~ Sepal.Width, data=iris)
```

The variable before the ~ will be plotted on the y-axis, and the variable after the ~ will be plotted on the x-axis.



Probability Distributions

Common discrete and continuous distributions are natively supported in R; each is associated with 4 functions useful for sampling and determining probabilities.

Function prefix	Use
d	Gives values of the density function of the given distribution
p	Computes cumulative probabilities of the given distribution
q	Computes quantiles of the given distribution
r	Takes random samples from the given distribution

Function family	Distribution
norm	Normal
binom	Binomial
unif	(Continuous) uniform
t	t
gamma	Gamma
pois	Poisson
exp	Exponential

Any combination of **function prefix** and **function family name** above yields a function for working with a probability distribution in R. The function family name specifies the distribution you want to work with, and the prefix specifies what action you want to take with regard to that distribution. The arguments of the functions depend on the parameters of each distribution. Type `?function_name` to see all the required arguments in each case.

```
> # For Z ~ N(0,1)
> pnorm(1) # Pr (Z <= 1)
[1] 0.8413447
> qnorm(0.95) # z such that Pr(Z <= z) = 0.95
[1] 1.644854
> # For T ~ t(df = 10)
> qt(0.99, df = 10) # t such that Pr(T <= t) = 0.99
[1] 2.763769
```

Sampling from a Vector

In addition to sampling from parametric distributions, we can also sample from a vector of values `x`. We can create a sample of size `n` from `x` **without replacement** using the command:

```
> sample(x, size=n, replace=FALSE)
```

If replacement is desired, change `FALSE` to `TRUE`.

```
> x <- c("A", "B", "C")
> sample(x, 2, replace = FALSE)
[1] "C" "B"
>
> sample(x, 6, replace = TRUE)
[1] "A" "B" "B" "B" "C" "C"
>
> sample(1:10, 6, replace = FALSE)
[1] 6 5 3 2 9 4
```

Statistical Inference

The R functions `prop.test()` and `t.test()` yield hypothesis test p-values and confidence intervals for one or two sample inferences on proportions and means, respectively. Both have optional arguments `alternative=` (taking on values in 'two.sided', 'less', 'greater' to specify the alternative hypothesis type, defaulting to 'two.sided') and `conf.level=` (taking on values between 0 and 1 to specify the desired confidence level, defaulting to 0.95).

INFERENCE ON PROPORTIONS

One sample:

Let `x` be the number of successes, `N` be the number of trials, and `p_null` be the probability of success under the null hypothesis.

```
> prop.test(x, n=N, p=p_null)
```

```
> # Suppose we observe 40 successes from 100 trials.
> # We want to test if the proportion is equal to 0.5.
> prop.test(40, n=100, p=0.5, alternative='two.sided', conf.level=0.90)

    1-sample proportions test with continuity correction

data:  40 out of 100, null probability 0.5
X-squared = 3.61, df = 1, p-value = 0.05743
alternative hypothesis: true p is not equal to 0.5
90 percent confidence interval:
 0.3183752 0.4872158
sample estimates:
  p
0.4
```

more➤

Two samples:

We can similarly use the `prop.test()` function to test for equality of two population proportions. Let `x` and `y` be the numbers of successes, and let `N` and `M` be the sample sizes for the first and second samples, respectively.

```
> prop.test(c(x,y), n = c(N, M))
```

`prop.test()` can also be run on a 2×2 table that counts outcomes of success/failure by a grouping variable.

A chi-squared test for independence can be run on contingency tables (and is particularly useful for comparisons of more than 2 outcomes/groups).

```
> # Sample 1: 30 successes out of 100 trials
> # Sample 2: 40 successes out of 90 trials
> # Are the population proportions of successes equal?
> prop.test(c(30, 40), n=c(100, 90), conf.level = 0.99)

    2-sample test for equality of proportions with continuity correction

data:  c(30, 40) out of c(100, 90)
X-squared = 3.6493, df = 1, p-value = 0.05609
alternative hypothesis: two.sided
99 percent confidence interval:
 -0.33426521  0.04537633
sample estimates:
   prop 1    prop 2
0.3000000 0.4444444
```

```
> # Does the proportion of flowers with sepal width > 2.7 differ among
> # versicolors and virginicas?
> tbl <- table(iris$Species, iris$Sepal.Width > 2.7)[-1,]
> tbl

           FALSE TRUE
  versicolor    21   29
  virginica     11   39
> prop.test(tbl)

    2-sample test for equality of proportions with continuity correction

data:  tbl
X-squared = 3.7224, df = 1, p-value = 0.05369
alternative hypothesis: two.sided
95 percent confidence interval:
 0.001395761 0.398604239
sample estimates:
prop 1 prop 2
  0.42   0.22

> chisq.test(tbl)

    Pearson's Chi-squared test with Yates' continuity correction

data:  tbl
X-squared = 3.7224, df = 1, p-value = 0.05369
```

INFERENCE ON MEANS

One sample:

If `x` is the vector of our observations and `mu_null` is the value of the mean under the null hypothesis, we can run a 2-sided hypothesis test and obtain the associated 95% confidence interval using:

```
> t.test(x, mu=mu_null)
```

```
> # Is the population mean sepal width of all setosas higher than 3.5?
> t.test(iris$Sepal.Width[iris$Species == "setosa"], mu=3.5,
alternative="greater")

    One Sample t-test

data:  iris$Sepal.Width[iris$Species == "setosa"]
t = -1.3431, df = 49, p-value = 0.9073
alternative hypothesis: true mean is greater than 3.5
95 percent confidence interval:
 3.338124      Inf
sample estimates:
mean of x
    3.428
```

Two samples:

With two samples, let `x` and `y` be vectors containing the observations from the first and second samples, respectively. The two population means can be compared using:

```
> t.test(x, y)
```

Alternatively, if a data frame `df` contains one column with `values` from both samples and one column with the sample `labels`, it is possible to use the following:

```
> t.test(values ~ labels, data=df)
```

By default, `t.test()` assumes that the variances of the two populations are *unequal*. This choice can be overridden via the optional argument `varequal=TRUE`.

```
> # Are the mean sepal widths the same between all versicolors and
> # virginicas?
> iris2 <- iris[iris$Species %in% c("versicolor", "virginica"),]
> t.test(Sepal.Width ~ Species, data=iris2)

    Welch Two Sample t-test

data:  Sepal.Width by Species
t = -3.2058, df = 97.927, p-value = 0.001819
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.33028364 -0.07771636
sample estimates:
mean in group versicolor  mean in group virginica
                   2.770                   2.974
```

Linear Regression

Suppose that we have two vectors of observations, `x` and `y`. An ordinary least squares line that predicts `y` using `x` can be fit using:

```
> lm(y~x)
```

If there is more than one explanatory variable to be included, say, `x1`, `x2`, and `x3`, the additional explanatory variables can be added to the right-hand side of the ~ using a + sign.

```
> lm(y ~ x1 + x2 + x3)
```

```
> lm.out <- lm(Sepal.Length ~ Sepal.Width, data=iris)
> lm.out

Call:
lm(formula = Sepal.Length ~ Sepal.Width, data = iris)

Coefficients:
(Intercept)  Sepal.Width
     6.5262      -0.2234

> summary(lm.out)

Call:
lm(formula = Sepal.Length ~ Sepal.Width, data = iris)

Residuals:
    Min      1Q  Median      3Q     Max
-1.5561 -0.6333 -0.1120  0.5579  2.2226

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   6.5262     0.4789   13.63   <2e-16 ***
Sepal.Width  -0.2234     0.1551   -1.44    0.152
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.8251 on 148 degrees of freedom
Multiple R-squared:  0.01382, Adjusted R-squared:  0.007159
F-statistic: 2.074 on 1 and 148 DF,  p-value: 0.1519
```

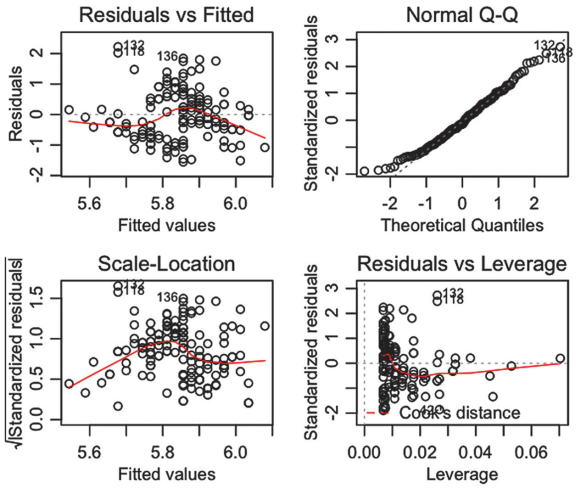
If `y`, `x1`, `x2`, and `x3`, are all columns in the `df` data frame, we can use the command:

```
> lm(y ~ x1 + x2 + x3, data=df)
```

When **all** columns in `df` other than `y` are intended to serve as predictors, a convenient shorthand is given by:

```
> lm(y ~ ., data=df)
```

It is useful to store the outputs of `lm()` into an object so that further useful insights can be extracted. For example, running `summary()` on the `lm` object yields a detailed regression summary, including R^2, a five-number summary of the residuals, and p-values describing the significance of each predictor. A series of regression diagnostics can be examined visually by running `plot()` on the `lm` object.



Common Pitfalls and Tips

- When specifying logical values `TRUE` or `FALSE`, for example, as arguments to functions, do not include quotes. "`TRUE`" is a character string and not a logical value.
- When entering a fresh command, ensure that the console prompt begins with '>'. If instead you see '+', R is waiting for you to complete an earlier command. Hit 'Esc' to start afresh.
- If you have questions about a function described in this card, you can access its help page by entering `?function_name` (e.g. `?dnorm`).

Beyond the Basics

R is a full-fledged scripting language that allows you to write complex programs that handle control statements [e.g., see `if()`] and manage tasks with repetition [e.g., see `for()` and `while()` loops].

A list of frequently used R functions is found here:
https://cran.r-project.org/doc/contrib/Short-refcard.pdf

You can install additional user-authored packages to extend the basic capabilities of base R by running `install.packages("package_name")`. A directory of available packages can be found at:
https://cran.r-project.org/web/packages/