Introduction to R and Rstudio

Andrew Raim

2025-02-22

Overview

Introduction

- We will try to get a sense of using R for data analysis, graphics, and programming, and how to use the Rstudio IDE.
- There are two parts.
    1. A walkthrough based on a series of R Tutorials from the Center for Interdisciplinary Research and Consulting (CIRC) at UMBC.
    2. Additional examples of basic programming.

What is R?

- The R Project for Statistical Computing
- A software system including a programming language.
  - Object-oriented. Functions are also objects.
  - Intended for data analysis, graphics, and quick prototyping of algorithms.
  - Good performance is possible, but simplicity is the priority.
- Developed by Ross Ihaka and Robert Gentleman at the University of Aukland in 1995 (Ihaka and Gentleman 1996).
- Free and open source.
- Based on the commercial software S/S-Plus.
- Code base is maintained by the R Development Core Team.

What is R?

- Cross platform: Windows, Mac, and Linux.
  - All platforms can be used through a terminal.
  - Windows version comes with a simple GUI.
- Huge selection of contributed packages from the user community.
  - The Comprehensive R Archive Network (CRAN)
  - Bioconductor
  - Github and other repositories
- You can make your own packages and share them with colleagues, collaborate on them through Github, submit them to CRAN, etc.
- Popular especially with statistics and statistical learning community in academia and other research-oriented institutions.

Rstudio

What is Rstudio?

- A premiere integrated development environment (IDE) for R.
- Created by JJ Allaire in 2008 as an open source project and currently maintained by Posit Software.
- The base IDE is still open source. Enterprise products, cloud services, and technical support are paid.
- Cross platform: Windows, Mac, and Linux.

Getting Started

Getting started with Rstudio

- Where to download and how to install.
- Interface of Rstudio.
- Closing Rstudio and the `q()` command.
- The Help panel and the function `rnorm`.
- The Packages panel.
  - Let's consider installing the COMPoissonReg package.
  - Install package from CRAN.
  - Load package into our session with `library`.
  - Search for package documentation.
  - View manual page for the function `rcmp`.
  - View the vignette.

A First Script

- Draw a sample $x_1, \ldots, x_n$ from $N(0, 1)$ and plot a histogram.

```
x = rnorm(100)
hist(x)
```

- Let's try the following.
  - Working in the script panel.
  - Assignment operators `=` and `<-`.
  - Varible names are case-sensitive.
  - Running the script
  - Results: output in console, figure in Plots panel, objects in Environment panel, command saved to History panel
  - Pass some arguments to `rnorm`; they can be positional or labeled.
  - Set some plot arguments: `nclass = 10` and `col = "blue"`.
  - Save the script.

Importing Data

- Browse to the PROFIT dataset and examine it.
- Download the file to our computer.
- Use `getwd` and `setwd` to get and set the working directory to the location of the file.
- Note the use of forward and backward slashes in path.
- Use `read.table` to read the data from the saved file.

```
df = read.table("PROFIT.txt", sep = '\t', head = TRUE)
```

- Browse the data frame from the Environment panel.
- Type the data frame's name in the console to print it.
- Save the file as a CSV.

```
write.csv(df, file = "PROFIT.csv", row.names = FALSE)
```

- Examine the resulting CSV.

Basic Analysis

Exploratory Statistics

- Histogram of `PRO`.

```
hist(df$PRO)
```

- Scatterplot of `POPN` and `PRO`.

```
plot(df$POPN, df$PRO)
```

- Boxplot of `PRO` grouped by `COMMIS`.

```
boxplot(PRO ~ COMMIS, data = df)
```

- Statistical summary of data frame with `summary` function.

```
summary(df)
```

- Correlation between `PRO` and `POPN`.

```
corr = cor(df$PRO, df$POPN)
print(corr)
```

One Sample t-test

- Suppose observations from variable PRO are from a normal distribution with unknown mean μ and variance σ^2.
- Let us test $H_0 : \mu \leq \mu_0$ versus $H_1 : \mu > \mu_0$ using the value $\mu_0 = 900$.
- A t-test rejects H_0 when test statistic $t = \frac{\bar{x}-\mu_0}{s/\sqrt{n}}$ is larger than an appropriate critical value of the t-distribution.
- Use the `t.test` function in R to carry out the test.

```
out = t.test(PRO ~ 1, alternative = "greater", mu = 900, data = df)
print(out)
names(out)
```

- We can also compute the test manually.

```
mu0 = 900
n = length(x)
xbar = mean(x)
s = sd(x)
alpha = 0.05                              ## Significance level
tstat = (xbar - mu0) / (s / sqrt(n))      ## The t-statistic
qt(alpha, df = n-1, lower.tail = FALSE)   ## Critical value for one-sided test
pt(tstat, df = n-1, lower.tail = FALSE)   ## P-value for one-sided test
```

Two Sample t-test

- The variable `COMMIS` is binary with values `0` and `1`.
- Suppose observations of `PRO` with `COMMIS = 1` and `COMMIS = 0` are from normal distributions $\mathrm{N}(\mu_1, \sigma_1^2)$ and $\mathrm{N}(\mu_0, \sigma_0^2)$, respectively.
- Let us test $H_0 : \mu_0 \geq \mu_1$ versus $H_1 : \mu_0 < \mu_1$ with a two-sample t-test.
- Use the `t.test` function in R to carry out the test.

```
1 out = t.test(PRO ~ COMMIS, alternative = "less", data = df)
2 print(out)
```

- By default, it does not assume that $\sigma_0^2 = \sigma_1^2$. How can we change this?

Linear Regression

- Let us fit a linear regression using `PRO` as the outcome and `POPN` as the dependent variable.
- Use the `lm` function.

```
out = lm(PRO ~ POPN, data = df)
```

- There are a number of useful accessors.

```
print(out)    ## Print the result
summary(out)  ## Get a more detailed display
predict(out)  ## Predictions for POPN values in training data
resid(out)    ## Compute residuals
confint(out)  ## Confidence intervals for regressioncoefficients
```

- A few diagnostic plots.

```
plot(df$PRO, predict(out))      ## Observe versus fitted
plot(predict(out), resid(out))  ## Fitted versus residuals
```

- Here is a more complicated regression formula using `COMMIS` as a factor.

```
out2 = lm(PRO ~ POPN:as.factor(COMMIS), data = df)
summary(out2)
plot(df$PRO, predict(out2))       ## Observe versus fitted
plot(predict(out2), resid(out2))  ## Fitted versus residuals
```

Saving Our Work

- Saving a script.
- Saving a plot.
- The workspace.
  - List objects with `ls`.
  - Save workspace with `save` and `save.image`.
  - Load workspace with `load`.

Basic Programming

Vector Operations

- Vectors can be created in many ways.

```
x1 = c(1, 1, 2, 2, 3)
x2 = 1:5
x3 = numeric(5)
x4 = rnorm(5)
print(x1); print(x2); print(x3); print(x4)
```

```
[1] 1 1 2 2 3
```

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

```
[1] 0 0 0 0 0
```

```
[1]  1.5053836 -0.7975715 -1.0667554 -0.1516093 -0.5242349
```

- Get the length.

```
length(x1)
```

```
[1] 5
```

Vector Operations

- Here are some ways to subset vectors.

```
print(x1)
```

```
[1] 1 1 2 2 3
```

```
x1[1]
```

```
[1] 1
```

```
x1[1:3]
```

```
[1] 1 1 2
```

```
x1[x1 > 1]
```

```
[1] 2 2 3
```

```
idx = which(x1 > 1)
print(idx)
```

```
[1] 3 4 5
```

```
x1[idx]
```

```
[1] 2 2 3
```

Vector Operations

- Arithmetic on vectors.

```
1 x1 + 10
```

```
[1] 11 11 12 12 13
```

```
1 x1 * 10
```

```
[1] 10 10 20 20 30
```

```
1 x1^2
```

```
[1] 1 1 4 4 9
```

```
1 x1 + x2
```

```
[1] 2 3 5 6 8
```

```
1 x1 * x2
```

```
[1]  1  2  6  8 15
```

```
1 x1^x2
```

```
[1]   1   1   8  16 243
```

Vector Operations

- Many single-variable functions can be applied to vectors elementwise.

```
1 log(x1)
```

```
[1] 0.0000000 0.0000000 0.6931472 0.6931472 1.0986123
```

```
1 sqrt(x1)
```

```
[1] 1.000000 1.000000 1.414214 1.414214 1.732051
```

```
1 exp(x1)
```

```
[1]  2.718282  2.718282  7.389056  7.389056 20.085537
```

Vector Operations

- Some functions operate on the entire vector.

```
1 x = rnorm(100)
2 sum(x)
```

```
[1] 3.722163
```

```
1 mean(x)
```

```
[1] 0.03722163
```

```
1 sd(x)
```

```
[1] 0.9313191
```

```
1 quantile(x)
```

```
         0%         25%         50%         75%        100% 
-2.63661002 -0.62608006  0.05747426  0.53365145  2.24317119 
```

Matrix Operations

- Here is one way to create a matrix.

```
1 X = matrix(1:6, 2, 3)
2 print(X)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

- Get the dimensions.

```
1 dim(X)
```

```
[1] 2 3
```

```
1 nrow(X)
```

```
[1] 2
```

```
1 ncol(X)
```

```
[1] 3
```

Matrix Operations

- A few ways to index.

```
X[1,2]  ## Element on first row and second column
```

```
[1] 3
```

```
X[1,]   ## First row
```

```
[1] 1 3 5
```

```
X[,2:3] ## Second and third columns
```

```
     [,1] [,2]
[1,]    3    5
[2,]    4    6
```

Matrix Operations

- Some basic arithetic with matrices.

```
Y = matrix(seq(-3, 2), 2, 3)
X + 1
```

```
     [,1] [,2] [,3]
[1,]    2    4    6
[2,]    3    5    7
```

```
X * 2
```

```
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
```

```
X + Y
```

```
     [,1] [,2] [,3]
[1,]   -2    2    6
[2,]    0    4    8
```

```
X * Y
```

```
     [,1] [,2] [,3]
[1,]   -3   -3    5
[2,]   -4    0   12
```

Matrix Operations

- Many single-variable functions can be applied to matrices elementwise.

```
1 exp(X)
```

```
         [,1]     [,2]     [,3]
[1,] 2.718282 20.08554 148.4132
[2,] 7.389056 54.59815 403.4288
```

```
1 sqrt(X)
```

```
         [,1]     [,2]     [,3]
[1,] 1.000000 1.732051 2.236068
[2,] 1.414214 2.000000 2.449490
```

Matrix Operations

- Matrix multiplication has a special `%*%` operator.

```r
u = matrix(c(1, 2, 3), ncol = 1)
print(X)
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```r
print(u)
```

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

```r
X %*% u
```

```
     [,1]
[1,]   22
[2,]   28
```

Arrays

- We can also construct multidimensional arrays with more than two dimensions.

```
1 x = array(data = 1:24, dim = c(4,3,2))
2 print(x)
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

```
1 x[,2:3,1]
```

```
     [,1] [,2]
[1,]    5    9
[2,]    6   10
[3,]    7   11
[4,]    8   12
```

Data Frames

- We already encounted data frames. Let's create one ourselves.

```r
x = 1:5
y = c("a", "a", "a", "b", "b")
z = rnorm(5)
df = data.frame(subject = x, group = y, measurement = z)
print(df)
```

```
  subject group measurement
1       1     a  -1.3481371
2       2     a  -0.2665445
3       3     a  -0.1909869
4       4     b   2.2902795
5       5     b  -0.4084638
```

- Columns can be accessed by name.

```r
df$measurement
```

```
[1] -1.3481371 -0.2665445 -0.1909869  2.2902795 -0.4084638
```

```r
df[["measurement"]]
```

```
[1] -1.3481371 -0.2665445 -0.1909869  2.2902795 -0.4084638
```

Data Frames

- Elements can also be accessed like matrices

```
1 dim(df)
```

```
[1] 5 3
```

```
1 df[,3]
```

```
[1] -1.3481371 -0.2665445 -0.1909869  2.2902795 -0.4084638
```

```
1 df[1,]
```

```
  subject group measurement
1       1     a   -1.348137
```

```
1 df[1,3]
```

```
[1] -1.348137
```

Functions

- We can easily create our own functions.

```
f = function(x) { return(x + 1) }
z = 1:5
f(z)
```

```
[1] 2 3 4 5 6
```

- We can also create functions around objects in our environment.

```
g = function(x) { z + x }
g(numeric(5))
```

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

- Pass a function as an argument.

```
h = function(f) { f(z) }
h(sd)
```

```
[1] 1.581139
```

- Return a function as a return value.

```
l = function() { return(sd) }
f = l()
f(z)
```

Lists

- Lists provide a flexible way to compose other data structures.

```
res = list()
res[[1]] = c(1, 2, 3)
res[[2]] = 1:5
res[[3]] = function(x) { x + 1 }
print(res)
```

```
[[1]]
[1] 1 2 3

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

[[3]]
function (x)
{
    x + 1
}
```

```
length(res)
```

```
[1] 3
```

Lists

- Use double brackets to access element.

```
1 res[[2]]
```

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

- Use single brackets to access element as a list.

```
1 res[2]
```

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

Some Numerical Tools

- Use `sort` to sort a vector and `order` determine a permutation needed to put the elements into sorted order.

```
1 x = c(-1, -2, -3, 0, 3, 2, 1)
2 sort(x)
```

```
[1] -3 -2 -1  0  1  2  3
```

```
1 order(x)
```

```
[1] 3 2 1 4 7 6 5
```

Some Numerical Tools

- Use `solve` to solve linear equations of the form $AX = B$.

```
A = matrix(1, nrow = 3, ncol = 3)
diag(A) = 2
b = matrix(c(1, 1, 1), ncol = 1)
x = solve(A, b)
print(x)
```

```
     [,1]
[1,] 0.25
[2,] 0.25
[3,] 0.25
```

- Also use `solve` to compute the inverse of A.

```
Ainv = solve(A)
print(Ainv)
```

```
      [,1]  [,2]  [,3]
[1,]  0.75 -0.25 -0.25
[2,] -0.25  0.75 -0.25
[3,] -0.25 -0.25  0.75
```

```
A %*% Ainv
```

```
              [,1] [,2] [,3]
[1,]  1.000000e+00    0    0
[2,] -5.551115e-17    1    0
[3,] -1.110223e-16    0    1
```

Some Numerical Tools

- Use `eigen` to compute eigenvalues and eigenvectors of A.

```
1 eigen(A)
```

```
eigen() decomposition
$values
[1] 4 1 1

$vectors
           [,1]       [,2]       [,3]
[1,] -0.5773503  0.0000000  0.8164966
[2,] -0.5773503 -0.7071068 -0.4082483
[3,] -0.5773503  0.7071068 -0.4082483
```

Some Numerical Tools

- Use integrate to compute integrals of the form $\int_a^b f(x)dx$.

```
f = function(x) { 1 / x }
integrate(f, lower = 1, upper = 10)  ## Numerical result
```

```
2.302585 with absolute error < 4.1e-05
```

```
log(10)                              ## Analytical form
```

```
[1] 2.302585
```

Some Numerical Tools

- Use the `optimize` function to optimize a real-valued function f of one variable over a bounded interval. Minimization is the default behavior.

```
f = function(x) { (x - 0.5)^2 }
optimize(f, interval = c(-5, 3))
```

```
$minimum
[1] 0.5

$objective
[1] 1.232595e-32
```

Some Numerical Tools

- Use the `optim` function to optimize a real-valued function f of multiple variables over Euclidean space.

```
f = function(x) { sum((x - 0.5)^2) }
optim(par = c(0,0,0), f, method = "L-BFGS-B")
```

```
$par
[1] 0.5 0.5 0.5

$value
[1] 0

$counts
function gradient
       4        4

$convergence
[1] 0

$message
[1] "CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL"
```

Printing to the Console

- The `print` function is implemented for almost every kind of object to display it. We have already seen it - e.g., to display a list.
- The `cat` function is also useful to display text.

```
guest = "Andrew"
cat("Hello world from", guest, "\n")
```

```
Hello world from Andrew
```

- The `sprintf` function useful for creating formatted text.

```
x = 17
y = "February"
z = 2025
msg = sprintf("Day %d of month %s in the year %d", x, y, z)
cat(msg)
```

```
Day 17 of month February in the year 2025
```

Basic Control Flow

- Here is a for-loop.

```
for (i in 1:4) {
    cat("Starting iteration", i, "\n")
}
```

```
Starting iteration 1
Starting iteration 2
Starting iteration 3
Starting iteration 4
```

- Here is a while-loop.

```
i = 1
while (i <= 4) {
    cat("Starting iteration", i, "\n")
    i = i + 1
}
```

```
Starting iteration 1
Starting iteration 2
Starting iteration 3
Starting iteration 4
```

Basic Control Flow

- Here is an `if` statement.

```
x = 0
if (x > 0) {
    cat("Condition was true")
}
```

- Here is an `if-else` statement.

```
x = 0
if (x > 0) {
    cat("Condition was true")
} else {
    cat("Condition was false")
}
```

```
Condition was false
```

- Here is an `if-elseif-else` statement.

```
x = 2
if (x == 1) {
    cat("First condition was true")
} else if (x == 2) {
    cat("Second condition was true")
} else {
    cat("Something else was true")
}
```

Basic Control Flow

- There is a family of “apply” functions that can be used to apply a function to rows, columns, or elements of a matrix, elements in a list, and more.
- Sometimes this style of coding is preferred to loops because it can be written more succintly. (It can also lead to code which is more difficult to read)
- Here we apply the `mean` function to the columns of a matrix.

```
1 X = matrix(rnorm(15), 5, 3)
2 print(X)
```

```
           [,1]       [,2]         [,3]
[1,]  1.0239727  0.5973456 -0.858638197
[2,]  1.0052769  0.5081917 -0.301916092
[3,] -0.5610416 -1.3472357  0.389317484
[4,] -0.4243939 -0.2236822 -0.007366002
[5,] -0.2528479 -0.7463043  0.356593125
```

```
1 apply(X, 2, mean)
```

```
[1]  0.15819322 -0.24233697 -0.08440194
```

- Note that matrix operations are typically orders of magnitude faster than loops and apply statements. Matrix operations are implemented in underlying C libraries like BLAS.

Basic Control Flow

- The pipe operator `|>` can be used string together sequences of statements. It can help with readability of code.

```
mtcars |> subset(cyl == "4") |> head(4)
```

```
            mpg cyl  disp hp drat   wt  qsec vs am gear carb
Datsun 710 22.8   4 108.0 93 3.85 2.32 18.61  1  1    4    1
Merc 240D  24.4   4 146.7 62 3.69 3.19 20.00  1  0    4    2
Merc 230   22.8   4 140.8 95 3.92 3.15 22.90  1  0    4    2
Fiat 128   32.4   4  78.7 66 4.08 2.20 19.47  1  1    4    1
```

- By default, the statement `x |> f(y)` passes `x`as the first argument of `f` and `y` is taken to be the second argument. We can use the _ placeholder to pipe to an argument other than the first one. To do this, we have to use labeled arguments.

```
f = function(x, y) { x^y }
z = 1:5
z |> f(2)         ## Default: pipe z into first argument of f
```

```
[1]  1  4  9 16 25
```

```
z |> f(2, y = _)  ## With placeholder, pipe z into second argument of f
```

```
[1]  2  4  8 16 32
```

Exceptions

- R functions throw errors when something goes wrong. Here is an example where we try to invert a singular matrix: a 2×2 with all zeros.

```
X = diag(x = 0, 2, 2)
solve(X)
```

```
Error in solve.default(X): Lapack routine dgesv: system is exactly singular: U[1,1] = 0
```

- If this occurs within a program, the entire program will halt. However, we can catch the error and handle (or ignore) it; one way is with the `tryCatch` function.

```
out = logical(5)
z = seq(0, 1, length.out = 5)
for (i in 1:5) {
    X = diag(x = z[i], 2, 2)

    out[i] = tryCatch({
        solve(X)
        TRUE                ## Produce TRUE if solve completes
    }, error = function(e) {
        FALSE               ## Produce FALSE if we catch an error from solve
    })
}
print(out)
```

```
[1] FALSE  TRUE  TRUE  TRUE  TRUE
```

Python Integration

- The `reticulate` package provides some interoperability with Python. See its documentation for more details.
- Recent versions of Rstudio also support running Python code.
- Here is a quick Python snippet to try.

```
guest = "Andrew"
print(f"Hello world from {guest}")
```

Conclusions

Further Exploration

- Tidyverse: family of packages for improved data wrangling, plotting, and other core data science tasks.
- Quarto: documentation system that includes markdown, executable code, mathematics, and more.
- Rcpp: framework to integrate R with C/C++ code.
- Stan: Bayesian modeling system that integrates with R and Rstudio.
- A few of the many books with contents available online.
  - R for Data Science: on the use of Tidyverse.
  - R Packages: on preparing R packages.
  - Advanced R: on advanced R programming.

Going Further

- Current developments and applications in R.
  - Journal of Statistical Software
  - R Journal
  - UseR!: an annual conference for the R community.
- Some alternative, portable, open source platforms for scientific computing.
  - Python is a general purpose language which is especially popular in machine learning. It also appears to be the most popular programming language overall, according to this article.
  - Julia is designed to produce fast executable programs. Has a dedicated community and a large selection of contributed packages. See Bezanson et al. (2017).

References

Bezanson, Jeff, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. “Julia: A Fresh Approach to Numerical Computing.” *SIAM Review* 59 (1): 65–98. https://doi.org/10.1137/141000671.

Ihaka, Ross, and Robert Gentleman. 1996. “R: A Language for Data Analysis and Graphics.” *Journal of Computational and Graphical Statistics* 5 (3): 299–314. https://doi.org/10.1080/10618600.1996.10474713.