BMI 217: R Tutorial

Diego Calderon, Peyton Greenside, and Daniel Kim

January 6, 2016

Contents

Getting started

The R programming language is widely-used in the bioinformatics field, therefore we will primarily use R as the coding language of this class. It is an excellent language for first-pass data exploration and more rigorous statistical analysis. The power of R comes from an emphasis on popular, easy-to-use, open-source packages easily accessible in centralized repositories. At the moment of writing this tutorial there are 7732 available packages on CRAN (the largest R package repository) covering a huge range of fields

As you will learn, R has its disadvantages: it is not the fastest language, the syntax can seem clunky especially for programmers with previous experience in python or C, and it lacks an agreed upon system for object oriented programming. However for our purposes, the large and active community that provide quality R software make up for these shortcomings.

This document aims to introduce the reader to R programming, and help develop skills necessary for self-guided R study and analysis of data.

Installing R Download the system appropriate R version from the CRAN website (cran.r-project.org), and run the file. The default settings are acceptable.

Installing Rstudio (optional) We recommend downloading and installing Rstudio for writing R code. Rstudio is an Integrated Development Environment (IDE) that helps with code organization and efficient scripting. It is especially useful for beginner programmers. Download the Rstudio installer from the download page and run the file. While using Rstudio for this class is optional, the rest of this tutorial will assume it is installed.

General notes

R is interpreted (no compilation step), dynamically typed (no need to declare a type), and imperative (statements change a program's state) for most of what we will be doing. R scripts are saved with a `.R` extension. Run an R script from the command line using the command `Rscript`, however we recommend opening up an R script in Rstudio and executing it line by line (or a highlighted portion of code) with **Command-Return**. In addition, with an R terminal running (type `r` in the Terminal or use the bottom left box in the Rstudio window), you can use the prompt as a calculator or type any valid R commands.

```
248 + 21 - 1 - 8 * 4 / 2
```

```
## [1] 252
```

To assign elements to variables use `<-` or `=`. R commands are separated by either a semicolon `;` or a newline. Lines with `#` in front are comments.

```
# this will not be evaluated because it is a comment
x <- 2; y <- 4
x + y
```

```
## [1] 6
```

Other generally useful functions are `ls()` which show which variables are assigned data in the current programming environment (in Rstudio this information is visible in the top right box under the "Environment" tab) and `rm()` to remove variables from memory.

For more information on specific R functions, call that function with a question mark at the beginning (`?`). For example, `?ls()` will display the documentation on the function used to list objects in memory.

Common data structures

We will briefly cover the most used data structures. There are vectors that contain elements of the same type in a single dimension. Factors contain elements but are only a finite number of values. Matrices also contain a single type of data but are indexed by two dimensions. Most importantly there are lists that contain data with heterogenous type and length, and a special type of list called a data frame, where each element has the same length. For a more detailed coverage of data structures see Advanced R or the R documentation.

Vectors

The most basic structure is the vector and it consists of an ordered collection of data. Usually you will see Numeric (reals or doubles), Character, Integer, or Boolean (logical) vectors. An element by itself is considered a vector of length one. To determine whether a variable is a specific type use the function `is.type()`, where type is replaced by the type of interest (e.g., `is.numeric()`). You can create vectors with `:` for numbers, `seq()` for a sequence, `rep()` to repeat elements or the data type's respective construction function

(`numeric()`, `character()`, etc). Aggregate vectors with the function `c()`. To access elements of a vector either select by integer or logical index using `[]`, or by symbolic name `x['first']` (after assigning names to the vector with the `names()` function). It is **important** to note that R starts indexing at 1 and **not** 0. Elements of vectors can be modified by accessing elements and assigning them values, again with `<-`. Use `length()` to identify the number of elements in a vetor. Sorting vectors can be done with the `sort()` or `order()` function.

```
x <- 1:3; y <- seq(1, 3, 1)
z <- c(x, y)
class(z) # numeric
is.numeric(z) # TRUE
z[1] # 1
z[1] <- 20 # modify element
z[1] # now 20
names(z) <- c('first', 'second', 'third',
              'fourth', 'fifth', 'sixth')
z['first'] # still 20
length(z) # should be 6
sort(z) # puts 20 at end
z[order(z)] # alternative
```

Factors

Factors are similar to vectors, but are used to describe items that can have a finite number of values i.e., categorical. They are implemented using an integer array to specify the actual levels and a second array of names that are mapped to the integers. **For our purposes, you will probably avoid using factors.** In most cases we recommend converting factors to character vectors using `as.character()`. More generally, use the function `as.type()` to convert a data structure to another type.

Matrices

Matrices are also like vectors, however they are indexed by two numbers. Create matrices with the `matrix()` function. The function `dim()` returns the dimensions of a matrix. Transpose a matrix with `t()`. Aggregate matrices by row with `rbind()` or by column with `cbind()`. Accessing and modifying elements of a matrix is similar to how it is done in vectors – use `[]` to access elements and `<-` to modify them. Symbolic names can be assigned for rows with `rownames()` and columns with `colnames()`, so you can reference elements by symbolic names.

```
m1 <- matrix(1, nrow=3, ncol=3) # create 3 by 3 matrix of 1
m2 <- matrix(2, nrow=3, ncol=3) # create 3 by 3 matrix of 2
m <- rbind(m1, m2)
m <- cbind(m, m)
dim(m) # now 6 by 6
t(m) # get transpose of m
m[1,] # select first row
m[, 1] # select first column
m1[1, ] <- 5 # change all first row to ones
rownames(m1) <- c('first', 'second', 'third')
m1['first',] # select with symbolic name
```

Lists and data frames

Think of a list like a hash table, or dictionary in python. They contain a collection of R objects, and unlike matrices or vectors, can contain data of heterogenous type and length. Create lists with the `list()` function (passing in "name = data" pairs as arguments) and transform a list into a vector using `unlist()`. Lists are slightly more complex in terms of referencing data elements. You can access an element of a list by number using `[[]]` (**note** the double bracket), or by symbolic name with `$`. Assign symbolic names with the `names()` function. Unlike vectors, one may have nested lists.

```
# making new lists, note heterogenous data and length, and nested list
x <- list(a = c(1,2,3), b = c('d', 'e', 'f'),
          c = "foo", d = list(e = 1:4, f = 'bar'))
names(x)
x[[1]] # index by number
x[['a']] # or by symbolic name
x$a # an alternative to previous line
unlist(x) # gets vector version of the list x
```

Data frames are lists with class `"data.frame"`. Think of them as tables with heterogenous column types, but each column must have the same length. Create data frames with `data.frame()` (passing in "name = data" pairs as arguments) and access elements either by integer index `[,]` or column name `$`. As in matrices, combine data frames by row with `rbind()` and column with `cbind()`. A common operation performed on multiple data.frames is the `merge()` (also known as join) function. Get number of rows and columns with `nrow()` and `ncol()`. Unlike lists, one may not have nested data frames.

```
# adapted from ?merge example
# create author data frame
authors <- data.frame(surname = c("Tukey", "Venables", "Tierney", "Ripley", "McNeil"),
                      nationality = c("US", "Australia", "US", "UK", "Australia"),
                      deceased = c("yes", rep("no", 4)), stringsAsFactors = F)
authors[1:3, 1:2] # index by number
authors$surname # or by name with $

# We can sort by rows in a data frame
authors[order(authors$surname),]

# create books data frame
books <- data.frame(name = c("Tukey", "Venables", "Tierney",
                             "Ripley", "Ripley", "McNeil", "R Core"),
                    title = c("Exploratory Data Analysis",
                              "Modern Applied Statistics ...",
                              "LISP-STAT",
                              "Spatial Statistics", "Stochastic Simulation",
                              "Interactive Data Analysis",
                              "An Introduction to R"), stringsAsFactors = F)

# let's say you wanted to look up which books had authors that were deceased, use merge
merge(authors, books, by.x = "surname", by.y = "name")
```

Loops and conditional statments

The dominant loop statement is the for loop, and its syntax in R is similar to other popular programming languages. Another heavily-used family of loops are the `apply()` functions. They apply a function to each

element of a data structure and return an appropriate data structure. These may seem confusing at first, but it is important to be able to use them.

```
mat <- matrix(c(1,3,2,6,5,4), ncol = 2)
# initialize vector to hold row means
means <- rep(0, nrow(mat))
# for loops
for (i in 1:nrow(mat)) {
  means[i] <- mean(mat[i,])
}
# apply, "1" is by row "2" is by column
apply(mat, MARGIN=1, FUN=mean)
# ...lapply is for lists, sapply is more user-friendly.
# although many functions have previously implemented vectorized versions
rowMeans(mat)
```

Control statements are the familiar, `if` and `else`. The `if` keyword is followed by a statement, enclosed in `()`, that should evaluate to a logical type and then the body of code, enclosed in `{}`, that should be executed if the statement evaluated to `TRUE`.

```
5 > 2
if (5 > 2) print("5 is greater than 2")
i <- 5
if (i == 5) print("i is equal to 5")
i <- 6
if (i != 5) print("i is not equal to 5")
if (i > 10) {
  print("i is larger than 10")
} else{
  print("i is not larger than 10")
}
```

Functions

Writing functions to abstract away low-level details is an important practise to develop for writing modular code. Creating functions in R is similar to variable assignment however, we include the keyword `function` followed by parameter arguments enclosed in `()` and the main body of the function in `{}`. If you have auxiliary functions written in another file load them with `source()`. The last line of the body code will be returned by the function, altneratively use `return()`.

```
# writing a standard error function
stderr <- function(x) {
  sqrt(var(x)/length(x)) # returned value since last line
}
stderr(1:20) # run function on vector 1:20
```

Data IO

Use `read.table` to read a data file and create a resulting data frame (**note** set `stringsAsFactors` to `F` to prevent automatic conversion to factors). Use `write.table` to write data frame or matrix objects to disk (typically I set `quote` to `F`, `sep` to `\t`, and `row.names` to `F`). For saving or loading more complex data use `save` and `load`.

Commonly used R packages

To install R packages use the function `install.packages('ggplot2')`, replacing ggplot2 with any package you want to install (note that if it is your first time installing a package R might ask for a mirror, the choice is not immediately important). Once a package has been installed, load that package using `library(ggplot2)`.

ggplot2

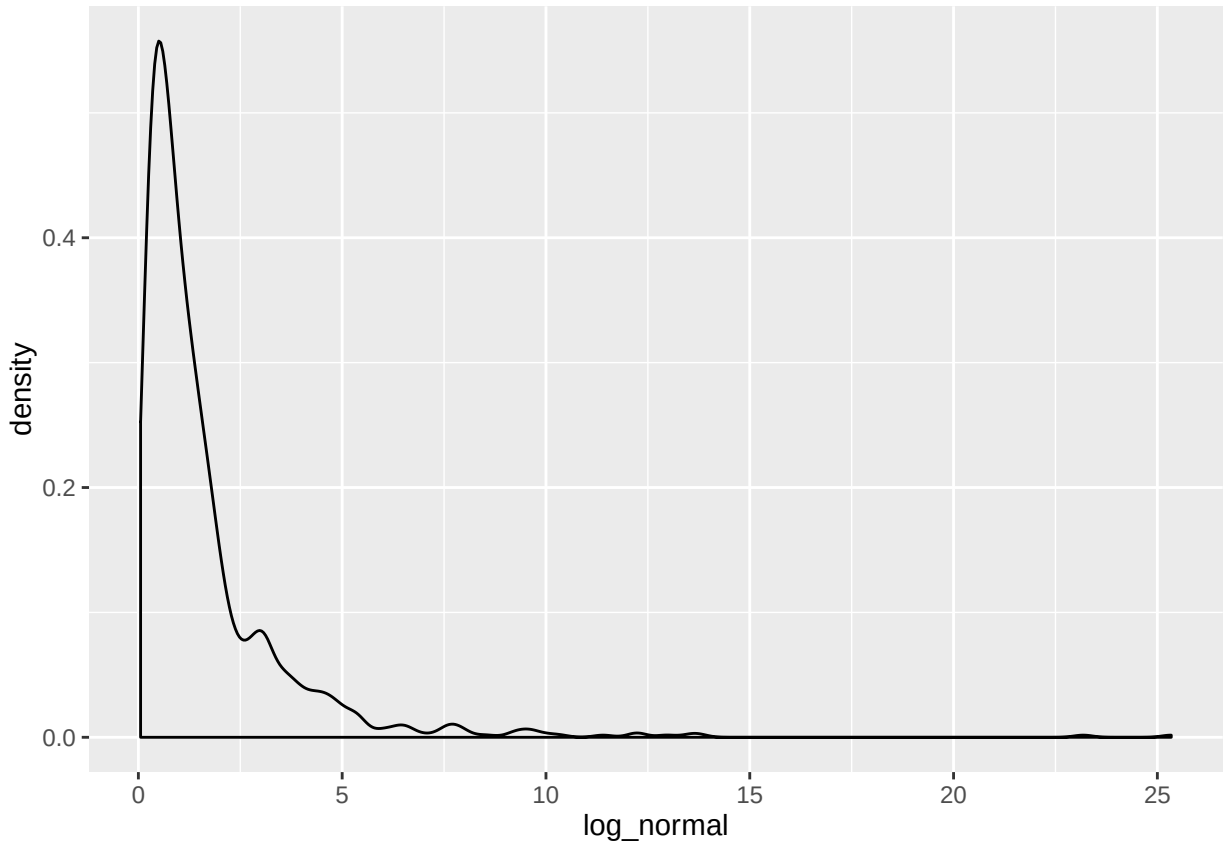
While base R has very strong graphic capabilities (`plot`, `barplot`, `hist`, `pie`), many prefer to create plots with ggplot2. Hadley Wickham wrote ggplot2 as an implementation of ideas from Leland Wilkinson's book *The Grammar of Graphics*. Even though this philosophy of plotting may seem strange, resulting plots are well worth the learning curve.

To make a plot, first pass in a data frame to the `ggplot()` function, along with an `aes()` object, which serves as an aesthetic mapping of data to visual properties of geoms. Finally, use the `+` operator to add different geom layers. After running this code, a plot should appear in the bottom right corner of the Rstudio window. To save it to disk click the export button and select desired options (alternatively use `ggsave()`). For more in-depth information and discussion of ggplot2 consult the excellent documentation, look up one of the many tutorials online, or talk to a TA.

```r
# load ggplot2
library(ggplot2)

# sample 1000 data points from a log-normal distribution
data <- data.frame(log_normal = rlnorm(1000))

# pass in data frame and aes (aesthetic mapping) + geom_layer
ggplot(data, aes(x = log_normal)) +
  geom_density() # creates a simple density plot
```



reshape

Another small but very useful package is **reshape**. Using the `melt()` function we can transform a data frame into a long-form data frame that is easier to use with ggplot.

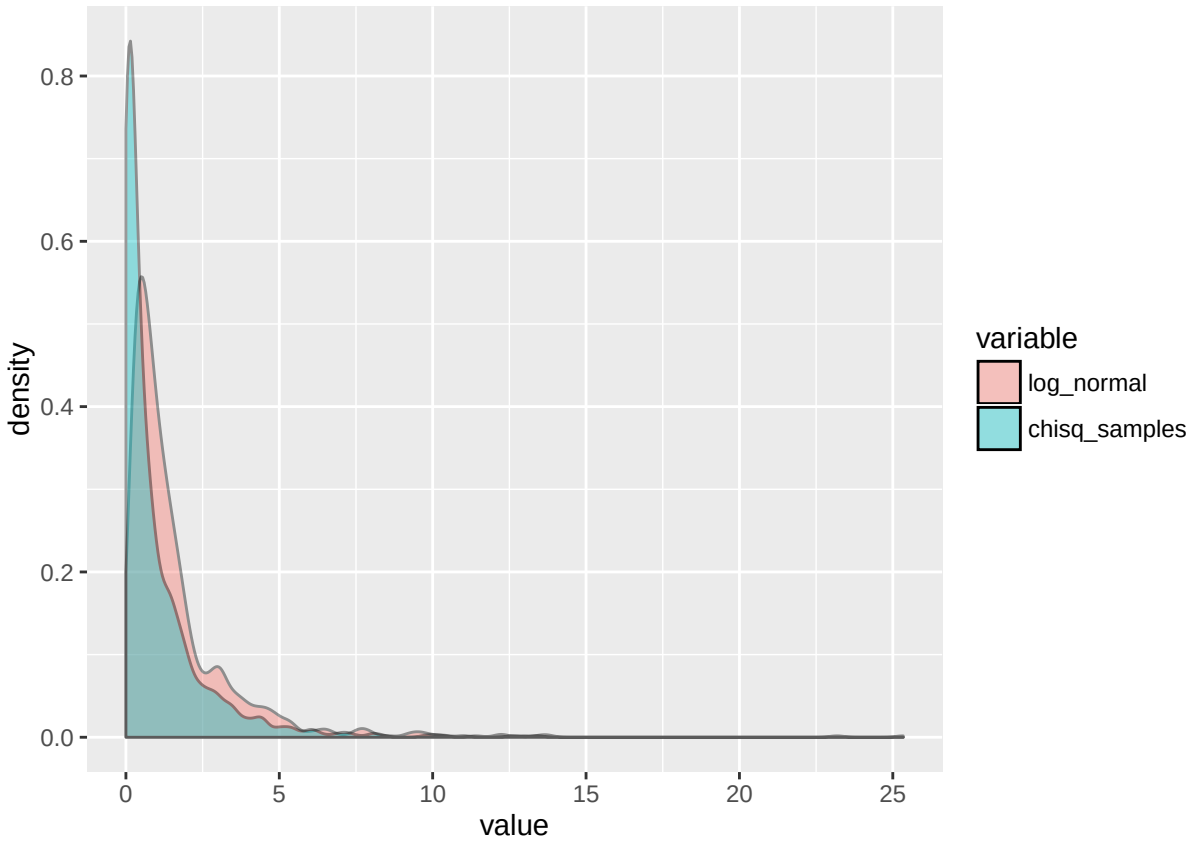
```
# load reshape
library(reshape)

# add more data sampled from chisq distribution
data$chisq_samples <- rchisq(1000, df = 1)

# melt data so value is the sample and the name is the variable
melted_data <- melt(data, id.vars = 0)

# explore long-form data
head(melted_data)
head(melted_data[melted_data$variable == 'chisq',])

# melt data so value is the sample and the name is the variable
# note that in aes fill is set to variable this is why we see different colors
ggplot(melted_data, aes(x = value, fill = variable)) +
  geom_density(alpha = 0.4) # alpha is the opacity
```



Other recommended R packages

Listed below are other R packages that we tend to use and recommend.

- dplyr - Eventually manipulating data frames becomes tedious, dplyr cleans up a lot of this kind of code.
- glmnet - Efficient functions for fitting lasso or elastic-net regularization regression models.
- rstan - R interface to Stan, which is a probabilistic programming language used for Bayesian statistical inference.
- GEOquery - Used to get microarray data from NCBI Gene Expression Omnibus (GEO).
- affy - Includes functions for microarray analysis.
- Rcpp - Allows R to integrate C++ code. Used for computation tasks that require faster than R-like speeds.
- Talk to the TAs if you have a specific need and we will help you find something.

Example excercises (optional)

- Look up the difference between the two assignment operators, `<-` and `=`.
- Explore operations that can be performed on vectors and matrices.
  - for example: `%*%`, `%in%`, `<`, `==`, `*`
- Look up the difference between accessing a list with `[]` versus `[[]]`.
- Write a function that prints "hello world" with the `print` function.
- Practise making plots with ggplot2 (and melt).

  - Load an example data.frame containing flower data using `data("iris")`.
  - Make a dot plot (`geom_point()`) comparing petal length by petal width and add a linear fit (`geom_smooth()`).
  - Make a boxplot of different petal widths for various species of flowers.
  - Play with different color and fill options.
- Use the function `lm()` to test whether Petal.Width is significantly predictive of Petal.Length.
- Look up how to install R packages from Bioconductor, install GEOquery, and download some data from GEO.

Other resources

- Google is your best friend for documentation and debugging.
- Many times googling an error will take you to stackoverflow where someone has asked the same question.
- CRAN is a great place to explore documentation and R packages.
- Many bioinformatics R packages are located on Bioconductor.
- The official R introduction.
- For reference on coding style use Google's R style guide.
- Hadley Wickham has written several extremely popular and useful R packages and a couple informative books that are worth a skim if you are interested in learning more about R.
  - Advanced R
  - R packages