Lab 1: R review

STA 602: Bayesian and Modern Statistics

Sep 05, 2022

Due: 12:00pm, Sep 06, 2022

Housekeeping

R/RStudio

You all should have R and RStudio installed on your computers by now. If you do not, first install the latest version of R here: https://cran.rstudio.com (remember to select the right installer for your operating system). Next, install the latest version of RStudio here: https://www.rstudio.com/products/rstudio/download/. Scroll down to the "Installers for Supported Platforms" section and find the right installer for your operating system.

R Markdown

You are required to use R Markdown to type up this lab report. If you do not already know how to use R markdown, refer to Sakai "Resources/Lab" folder for a very basic R Markdown template as well as a list of resources to help you learn how to use R markdown and LaTex.

Gradescope

You MUST submit both your .Rmd and .pdf files to Gradescope. Make sure to knit to pdf and not html; ask the TA about knitting to pdf if you cannot figure it out. Be sure to submit under the right assignment entry.

Getting started

This course will assume that you have some familiarity with the R programming language. However, to refresh your memory on some R key topics, this lab will go through some basic aspects of R functionality that will be important for future exercises.

As a first step, recall how to install new R packages using the R console. One very useful package for working with data is the `tidyverse` package. Try installing it with

```r
install.packages("tidyverse")
```

Two other packages we will encounter in future labs are `rstan` and `rstanarm`. Install these with the command

```
install.packages(c("rstan", "rstanarm"))
```

Reading and writing files

Depending on who you collaborate with, you may receive data in different forms. These might include Excel (.xls), text files (.txt), comma-separated-value files (.csv), or R DAT files (.dat). One common function we use to load data into the environment is the read.table() function. The call might look like `read.table(file = "file name")`. The file name is fully determined by where the data is located on your device, and where your current working directory is.

Sometimes, we may be able to access data from a website. Let's try to import data from Peter Hoff's "A First Course in Bayesian Statistical Methods":

```
# for the entire help file for read.table(), type ?read.table into your console
data <- read.table(file = url("http://www2.stat.duke.edu/~pdh10/FCBS/Exercises/azdiabetes.dat"))
head(data)
```

You will notice we successfully loaded the data (you should see it in the Environment panel). However, the first row of the dataset is clearly composed of the variable/column names. A quick fix sets the header parameter to TRUE to indicate that the first line of the file contains the names of the variables.

Exercise

1. Create a code chunk and set the header parameter to TRUE and print out the top rows of the table with `head()` as above.

```
data <- read.table(file = url("http://www2.stat.duke.edu/~pdh10/FCBS/Exercises/azdiabetes.dat"),
                   header = T)
head(data)
```

```
##   npreg glu bp skin bmi  ped age diabetes
## 1     5  86 68   28  30 0.36  24       No
## 2     7 195 70   33  25 0.16  55      Yes
## 3     5  77 82   41  36 0.16  35       No
## 4     0 165 76   43  48 0.26  26       No
## 5     0 107 60   25  26 0.13  23       No
## 6     5  97 76   27  36 0.38  52      Yes
```

Objects

R utilizes objects called data structures, which include numeric vectors, matrices, lists, and data frames. If you are ever unsure about a variable's type, you can run the command `str(<variable>)`.

We will frequently work with vectors and matrices. Vectors can be assigned using the `c()` function (the 'c' stands for 'combine'). We can apply element-wise arithmetic to numeric vectors as follows. Make sure that you understand the following operations!

```r
# create numeric vector of length 5
num <- c(1,2,3,4,5)

#the following are equivalent:
num + 1
num + c(1,1,1,1,1)

num + c(1,1) # why does this throw an error?

# multiplication
num2 <- num*2
num2*c(0,0,1,1,1)

# power
num2^2
```

Above, we created a numerical vector by listing out the elements 1,2,3,4,5. This works just fine if we only need a few elements. But what if we want all the integers between 1 and 100? We can use the `seq()` function to generate a numerical vector of values from the specified lower and upper bounds, at some regular interval:

```r
seq1 <- seq(from = 1, to = 100, by = 1) # the 'by' parameters determines the interval
seq1
```

Exercise

2. Generate a sequence of 100 equispaced real numbers from 0 to 1 and store it in a variable called `seq2`.

```r
seq2 <- seq(from = 0, to = 1, by = 1/100) # the 'by' parameters determines the interval
seq2
```

```
##   [1] 0.00 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
##  [16] 0.15 0.16 0.17 0.18 0.19 0.20 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29
##  [31] 0.30 0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.40 0.41 0.42 0.43 0.44
##  [46] 0.45 0.46 0.47 0.48 0.49 0.50 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59
##  [61] 0.60 0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69 0.70 0.71 0.72 0.73 0.74
##  [76] 0.75 0.76 0.77 0.78 0.79 0.80 0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.88 0.89
##  [91] 0.90 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99 1.00
```

Let's say we want to retrieve the i-th element in a vector. This is called indexing, and we do so using bracket notation. For example, if we want to retrieve the tenth element from seq1, then i=10:

```r
seq1[10]
```

If we want the first and last elements, we can index the positions as follows:

```
seq1[c(1,100)]
```

If we want every element except the 25th and 50th, we can easily use the syntax:

```
seq1[-c(25,50)]
```

Suppose we don't want the first element in seq1 to be 1, and would rather change it to 0. We can re-assign elements in numeric vectors by indexing the element we wish to re-assign, and specifying the replacement using `<-`:

```
seq1[1] <- 0
seq1
```

There are many helpful functions which operate on numerical vectors. We can easily calculate the mean, standard deviation, variance, length, and much more! Below are just some examples of functions which you may find useful in this course and moving forward in your statistical career:

```
seq3 <- seq(from = -3, to = 3, by = .5)
mean(seq3)
sd(seq3)
var(seq3)
length(seq3)
abs(seq3)
exp(seq3)
sqrt(seq3)
is.na(sqrt(seq3))
```

Exercise

3. Sort the entries in `seq3` from greatest to least.

```
seq3 <- seq(from = -3, to = 3, by = .5)
sort(seq3, decreasing = T)
```

```
##  [1]  3.0  2.5  2.0  1.5  1.0  0.5  0.0 -0.5 -1.0 -1.5 -2.0 -2.5 -3.0
```

Matrices

In this course we will often utilize matrices, which are objects where every row/column is itself a numerical vector. To create a matrix, we use the function matrix() which takes in the data elements, number of columns and rows, and arrangement as arguments. We can then perform matrix arithmetic and operations, such as addition, multiplication, transpose, inverse. For example:

```
mat1 <- matrix(data = seq(from = 1,to = 6, by =1), nrow = 3, ncol = 2, byrow = T)
mat2 <- matrix(data = rep(x = 2, times = 6), nrow = 3, ncol = 2)
mat3 <- mat1+mat2

#transpose of matrix: t()
t(mat3)

# matrix multiplication: %*%
# make sure dimensions agree!
dim(mat1); dim(mat2)
mat1 %*% t(mat3)

# inverse (if non-singular): solve()
mat4 <- matrix(data = c(1,2,3,4), nrow = 2, ncol = 2, byrow = F)
solve(mat4)

# obtain elements of main diagonal: diag()
diag(mat4)

# create a diagonal matrix: diag(x, nrow, ncol). Default is x=1, which creates an identity matrix
diag(4) # creates 4x4 identity matrix
diag(x = 2, nrow = 4) # creates 4x4 diagonal matrix with 2 on diagonal
```

Indexing matrices is similar to indexing vectors, but now that we are working in 2 dimensions, we need to specify the desired row and column. If we want the element in the i-th row and j-th column of mat4, then we use the syntax mat4[i,j]. If we want the entire i-th row, then we can leave the column index blank: mat4[i,] (and similarly for column).

```
mat4[1,1]
mat4[,2]
mat4[,2] <- c(0,0)
mat4
```

Lastly, suppose we have a large matrix and we wish to find the mean of every column. `apply()` is an extremely useful function which allows you to apply a specified function to an object, either row or column-wise. The function returns a vector or array of values:

```
# generate large matrix
mat5 <- matrix(seq(1,100,1), nrow = 4, ncol = 25, byrow = T)

# apply(X = object, MARGIN = 1 for rows or 2 for columns, FUN = function of choice)
# find mean of every column
apply(X = mat5, MARGIN = 2, FUN = mean)
```

Exercise

4. Find the variance of each row of `mat5`

```r
mat5 <- matrix(seq(1,100,1), nrow = 4, ncol = 25, byrow = T)
apply(X = mat5, MARGIN = 1, FUN = var)
```

```
## [1] 54 54 54 54
```

You can define your own functions and pass them into the FUN argument as well. For example, suppose we want to calculate the natural logarithm of the maximum element in a row/column. We can write a function which takes in a vector and returns the range. We then feed that function in as an argument into `apply()`:

```r
# create a function to calculate log of maximum for arbitrary x, and returns the value stored in ret
log_max <- function(x) {
  ret <- log(max(x))
  return(ret)
}

# find log of maximum for each column
apply(X = mat5, MARGIN = 2, FUN = log_max)
```

Data frames

An R data frame is an array, so the columns/variables can be of different types. If we use the `str()` function to determine the type of object the data we imported is, we notice that the 'data' objects is of type data frame, but the eight variables in the data frame are of different types (int, num, Factor). This differs from matrices, where the elements are solely numerical.

```r
str(data)
```

You may be accustomed to extracting a column of data using the dollar sign operation. We can also use tidyverse language to access/manipulate data frames. However, what is returned from the `select()` function is not a numeric vector but another data frame:

```r
# Extract blood pressure from data
bp1 <- data$bp
str(bp1)

# Select blood pressure from data
bp2 <- data %>% select(bp)
str(bp2)
```

In this course we will almost exclusively work with numerical data. Therefore, you may not need to foray into data frames. However, learning the tidyverse syntax can be very useful if you continue to work with R after this course.

Random number generation and distribution functions

Many aspects of Bayesian inference – and therefore aspects of this course – involve simulating random variables from well-known families of distributions. These include the discrete Bernoulli, Poisson, and Binomial families, as well as the continuous Gaussian, Gamma, and Beta distributions.

You can obtain samples from these and other distributions using commands like:

```
# Default is such that the first argument specifies the number of random samples you would like
X <- rnorm(10000, mean = 0, sd = sqrt(2))
Y <- rgamma(10000, shape = 1/2, rate = 1/2)
Z <- rpois(10000, lambda = 5)
```

You can also return the numerical values of the quantiles of these distributions (the inverse of the cumulative distribution function (CDF)) and their probability densities at desired values with slight variations to the base name of the distribution:

```
std_norm_qt <- qnorm(0.95) # For what value x will the CDF function of a N(0,1) R.V. return 0.95?
std_norm_cdf <- pnorm(-2) # What is the value of the CDF function of a N(0,1) R.V. at -2?
std_norm_dens <- dnorm(0.5) # What is the value of the PDF function of a N(0,1) R.V. at 0.5?
```

The distributions you will see in this class almost always come from families that are indexed by one or two parameters. For instance, in the example above, the object `X` contains samples from a Gaussian distribution with mean parameter $\mu = 0$ and variance parameter $\sigma^2 = 2$. Be sure to check the function documentation (with `?rnorm` or a similar command) to see exactly how these functions are parametrized. For instance, as we saw above, the `rnorm` function takes the standard deviation σ as its second argument, *not* the variance σ^2. Special care must also be taken to specify the shape or rate parameterization for the Gamma distribution.

Exercise

5. Generate 500 samples from a Beta distribution with shape parameter $[a, b] = [0.5, 0.5]$ and store the samples in a variable called `W`

```
set.seed(32507)
W <- rbeta(500, 0.5, 0.5)
head(W) # output is big hence not shown completely
```

```
## [1] 0.579 0.094 0.005 0.650 0.344 1.000
```

Plotting

When making plots in R, you have two main options: (1) the base R plotting function `plot` and (2) the package `ggplot2`.

The base R plotting functions are nice for quick, simple visualizations of data. Here are some examples:

```
set.seed(253)
norm_samples <- rnorm(10000)
#
par(mfrow = c(2, 2)) # Set the number of rows and columns for display panels
#
hist(norm_samples,
```

```r
     main = "Base R histogram",
     xlab = "x", ylab = "Count")
#
plot(x = norm_samples, y = pnorm(norm_samples),
     main = "Base R scatterplot",
     xlab = "x", ylab = "Phi(x)")
#
boxplot(norm_samples,
        main = "Base R boxplot",
        ylab = "x")
#
plot(density(norm_samples),
     main = "Base R density",
     xlab = "x", ylab = "Density")
```

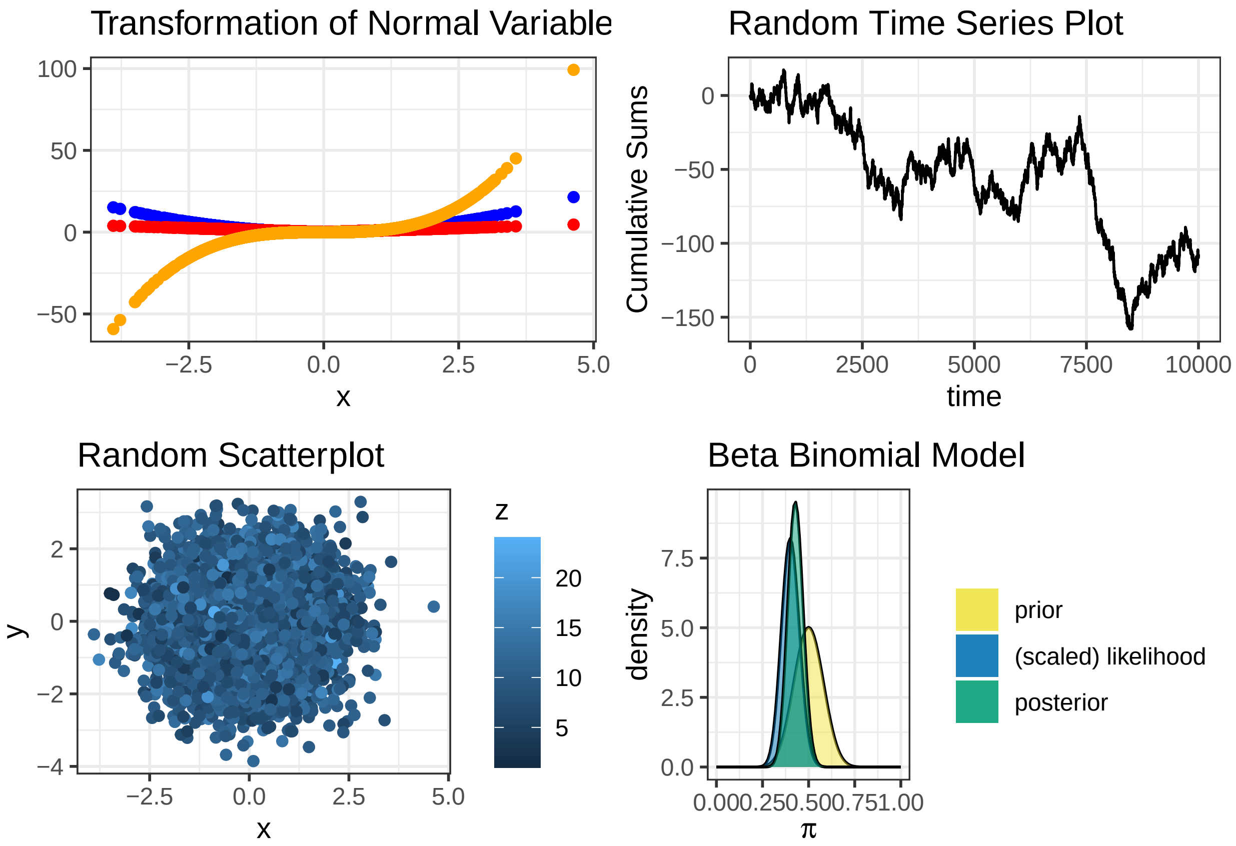
Plotting with `ggplot2` is generally a little bit easier when working with data in large tables. It has a gallery of built-in themes, and there are many extensions that make producing complicated visualizations relatively straightforward.

```r
norm_samples %>%
  data.frame(x = .) %>%
  ggplot2::ggplot() +
  geom_histogram(aes(x = x, y = ..density..),
                 fill = "#756bb1", colour = "white",
                 alpha = 0.5, bins = 30) +
  geom_density(aes(x = x), colour = "#756bb1") +
  geom_vline(aes(xintercept = std_norm_qt)) +
  geom_point(x = 0.5, y = std_norm_dens) +
  labs(x = "x", y = "Density", title = "ggplot density / histogram")
```

Exercise

6. Browse online resources (some below), or use code from above to make a few plots of your own.

```r
set.seed(253)
norm_samples <- rnorm(10000)

a <- norm_samples %>%
  data.frame(x = .) %>%
  mutate(`x^2` = x^2,`x^3` = x^3, `abs(x)` = abs(x)) %>%
  ggplot2::ggplot() +
  geom_point(aes(x,`x^2`),color = "blue") +
  geom_point(aes(x,`abs(x)`),color = "red") +
  geom_point(aes(x,`x^3`),color = "orange") +
  labs(title = "Transformation of Normal Variable", y = "") +
  theme_bw()

b<- norm_samples %>%
  data.frame(x = .) %>%
  mutate(`Cumulative Sums` = cumsum(x), time = 1:10000) %>%
  ggplot2::ggplot() + geom_line(aes(time,`Cumulative Sums`)) +
  labs(title = "Random Time Series Plot") +
  theme_bw()

set.seed(466)
norm_samples_2 <- rnorm(10000)
poission_samples_2 <- rpois(10000,10)

c <- data.frame(x = norm_samples, y = norm_samples_2, z = poission_samples_2) %>%
  ggplot2::ggplot() + geom_jitter(aes(x,y,color = z)) +
  labs(title = "Random Scatterplot") +
  theme_bw()

d <- bayesrules::plot_beta_binomial(alpha = 20, beta = 20, y = 40, n = 100) +
  labs(title = "Beta Binomial Model") +
  theme_bw()

grid.arrange(a, b, c, d, nrow = 2)
```



R tutorials and resources

For more information on the R programming language please refer to some or all of the following resources:

- R for Data Science
- ggplot2
- ColorBrewer

Grading

Total 10 points. 10 > Excellent; 8 > Good; 6 > Fair; 0 > Poor; NA if no submission.

Acknowledgement

This lab was created by Jordan Bryan and Becky Tang.