Introduction to R

Andreas Santucci
Institute for Computational and Mathematical Engineering
Stanford University
Summer 2018

Contents

1 Introduction

1.1 History

R is a powerful open-source tool for statistical computing. It's earliest origins date back to Bell Labs in the 70's via S (programming language). The motivation was for 'S' was to move away from calling Fortran routines and offer an interactive environment for statistical analysis. In the early 90's a "different implementation" of S by Ross I. and Robert G. started gaining traction.

1.2 Why R?

- R is excellent for *statistical analyses*.

  Methods are built in for (non)-linear regression modeling, time-series analysis, classification, clustering, and plotting. For these, we don't even need to import a package.

- R is *interactive*, and enables us to explore data and refine models iteratively.

  R is an *interpreted* language, i.e. we don't need to compile our programs into machine instructions before executing our code. We can simply enter code into the console and the result appears!

- R is *flexible*, allowing procedural, object-oriented, or even functional programming approaches.

  The practical implication is that we can run models and perform matrix arithmetic at a high level of abstraction, allowing us to think about the actual problem at hand.

- The active R development community makes the program *extensible*.

  Aside from the core R development team, we have Dirk Eddelbuettel on `Rcpp` and performance computing in R, Hadley Wickham who focuses on redesigning R code to be more user friendly via the `tidyverse`, and Matt Dowle leading the `data.table` development, just to name a few.

I have been greeted by many individuals who are surprised at what I have accomplished by programming in R. Unsurprisingly, those who are most aghast are typically lesser practiced in this language.

1.3 10k Foot Overview

How can we load, visualize, and model data? It's easy as π: try pasting the following into your console (you'll have to probably retype the `~` character that appears in the third line (linear model), since they tend not to paste over correctly from a `pdf` into a text editor).

```
webSite <- 'https://web.stanford.edu/~hastie/ElemStatLearn/datasets/ozone.data'
data  <- read.table(webSite, sep = '\t', header = TRUE)  # Load data into memory.
model <- lm(ozone ~ radiation, data)                     # Regress ozone onto radiation.
summary(model)                                  # Print summary model output to console.
```

We can type `head(data)` into our console to inspect the first few rows of our data, or for example `plot(data[, 1:4])` will plot the pairwise scatter relations for the first four columns in our data.

1.4 Course Outline

We'll start by setting up a development environment so that we can program in R comfortably. We'll then be equipped to learn how to store data in R and operate on it. We'll discuss fundamental data structures (like `data.frame` for tabular data) and control flow in R. Using these, we carry out an analysis. [1]

[1] If you're already familiar with programming in general, you may consider attempting some of our exercises in section 11.

2 Setting Up R

We start by downloading and installing the latest version of R on our machine. With this, we can start programming in R, albeit not in a friendly environment.

2.1 R Studio

I recommend starting with a popular interface for programming in R, R Studio. The interface consists of a *layout* of several windows:

- **Console:** This is where you type commands.
- **Editor:** This is where you write your code and save it to a text file.
- **Workspace/History:** What values exist in memory, and what's been executed historically.
- **Files/Plots/Packages/Help:** Allows graphical navigation of folders, display of visualizations, methods for installing and loading packages, and also a help utility.

2.2 Emacs Speaks Statistics

A more old-school, extensible approach relies on ESS. See documentation. Most notably, to start R we use `M-x R RET`. We can send a single line of code from a text file to our ESS process via `C-<RET>`, or alternatively we could use `C-c C-c` to run not an entire block or region of code and then step to the next.

2.3 Working Directory

If you're new to programming, this concept is an expected stumbling block. When we usually "use a computer", we navigate to a file or executable we want via a graphical interface and perhaps "double-click" to launch it; spare for launching the program, we typically don't care "where we are" in the filesystem. When we are programming, we have to learn to interact with files (and their filepaths) a *different* way.

A program operates out of some directory, where it defaults to reading/writing data to disk unless otherwise specified. If you have an R session open, we may type `getwd()` in the console to *get* our current *working directory*. We can also type `list.files()` to view the files and folders which are accessible from within our current working directory.

3 First Examples with R

3.1 As (an extensible) Calculator

R replaces our graphing calculators. Simply write an expression into the console and hit `<return>`. E.g.

```
10^2 + 36
```

```
## [1] 136
```

Exercise Compute the difference between 2018 and the year you graduated from your most recent educational program. Divide this by the difference between 2018 and the year you were born. Multiply the resulting value by 100 to determine the percentage of your life that you have been enjoying the fruits of your labor. You may find the use of parentheses helpful to disambiguate order of operations.

3.2 Variables

We can also give values a name. When we attach an identifier to a value, we realize a *variable*. E.g.

```
a <- 4
```

If you're using R-Studio, you'll now notice that `a` appears in the workspace window. We can also ask `R` what value `a` takes on, simply by typing `a` followed by `<return>` in the command window.

We can perform calculations on a variable, e.g.

```
a * 5

## [1] 20
```

If we specify a new value for `a`, we lose the old value.

```
a <- a + 10
a

## [1] 14
```

We can view all the objects in our workspace by typing `ls()`. We can remove all variables from R's memory by entering `rm(list=ls())`.[2]

Exercise Repeat the previous exercise, this time taking several steps to arrive at the final result. Assign each intermediate result to a variable, whose name can be one of your choosing (but is required to begin with a letter).

3.3 Scalars and Vectors

Numerical values are represented via scalars (single numbers, zero dimensional), vectors (a column of numbers, one dimensional), and matrices (a table of data, two dimensional). In our previous example, the `a` we defined was a *scalar*.

To define a *vector* of values, we use the `c()` command which stands for *concatenate*. E.g.

```
vals <- c(4, 7, 10)
```

We can also create a sequence of consecutive integers using the `:` operator.[3] E.g. inputting `0:9` into your console will return a length 10 vector of ordered digits.

3.4 (Using) Functions

What if we want to compute the average of the elements in `vals` above? We could manually type

```
(4 + 7 + 10) / 3
```

[2] This command should never be saved within an R-script, since it implies the script can never be run in the middle of any other workflow.

[3] If we want non-consecutive but equi-spaced values, we can use `seq`, e.g. `seq(0, pi, length.out = 5)` generates five evenly spaced values between zero and π.

but this would be error prone and impracticable for more interesting calculations. Common tasks are *automated* into *functions*: given input(s), they have a sequence of computations which yield an output. Some functions are available in base R, some are available in packages on CRAN, and if all else fails we can always write our own. To compute an arithmetic average, we simply type

```
mean(x = vals)

## [1] 7
```

Within the parentheses, we've specified the *function arguments* (or parameters). For the `mean` function, it requries an `x` vector for which we will compute an arithmetic average.

Exercise Compute the sum of `4, 5, 8, 11` by first combining the elements into a single vector and then using the function `sum`.

Exercise What's the sum of consecutive integers between 1 and 100 inclusive? Do you remember the formula for the sum of consecutive integers between 1 and N as a function of N? Hint: for any $n \in \mathbb{N}$, what is the result of the computation `1:n + n:1`? What does each value take on, and how many values are there?

Statistical Functions What's so great about R is that it has statistical functions built-in and easily accessible. E.g. if we type

```
rnorm(10)
```

we create a vector of ten draws from a standard normal distribution. If we run the same command again, we'll see ten *different* random numbers. To draw from a non-standard normal with mean $\mu = 10$ and standard deviation $\sigma = 3$, we may execute

```
rnorm(n = 10, mean = 10, sd = 3)

##  [1] 10.325045 13.317948  7.297843  9.087873 10.691250  9.299767 10.674679 14.651638 15.395
## [10]  7.381400
```

where here we've specified our arguments by name to disambiguate our intent (recommended). We remark that in R-Studio, if we simply type `rnorm(` within the command window, followed by pressing `TAB`, we will be prompted with the possible arguments to the function.[4]

Exercise Draw 100 random numbers from a normal distribution, assign the result to a variable `x`. Then, plot the result using `plot(x)`.

4 Help and Documentation

R has fantastic built-in documentation. Simply typing `help(<function_name>)` will pull up a help-page for a particular function, e.g. `help(rnorm)`. A prefix short hand for `help()` is the `?` operator, e.g. try inputting `?rnorm` into console.

[4] Emacs and ESS have similar functionality built-in.

5 Scripts

One advantage of R is in *reproducibility*. I.e. instead of using point-and-click GUI's to navigate spreadsheet calculations, for example, we can embed these instructions into code which is reusable. We store a sequence of R expressions in `.R` files which we call *scripts*. It's possible to run the entirety of another script from within R by using the `source()` function.

Exercise Make a file titled `firstRScript.R` in your *current* working directory. Request that R generate a hundred random numbers uniformly at random from an interval $[a, b]$ for any $a, b \in \mathbb{R}$ of your choosing. *Hint:* see `?runif` for help on how to draw values uniformly at random from an interval. Assign the resulting draws to a variable, and plot the result using `plot()`. Then, run this script a couple times, using `source()` from within an R console.

6 Essential Data Structures for Statistics

6.1 Vector (Operations)

We've seen how we can create vectors using `c()`, `:`, and functions like `seq` or `rnorm`. We can also *index into* these vectors via our subscript operator `‘[‘`, or *add* them together.

```
u <- c(1, 2, 4, 8, 16)
v <- seq(from = 0, to = 1, length.out = length(u))  # (0, 1/4, 1/2, 3/4, 1)

print(u)

## [1]  1  2  4  8 16

sum(u)

## [1] 31

u[3]        # R is 1-indexed.

## [1] 4

u[3] <- 0   # We may access and overwrite individual elems.
rbind(u, v) # Row-bind two vectors.

##   [,1] [,2] [,3] [,4] [,5]
## u    1 2.00  0.0 8.00   16
## v    0 0.25  0.5 0.75    1

u + v       # Vector addition is performed  element-wise.

## [1]  1.00  2.25  0.50  8.75 17.00
```

The last expression demonstrates vector arithmetic behaves in the obvious way, i.e. element-wise.

6.2 Matrices

A matrix (in R) is just a vector with attributes (number of rows, number of columns, and a corresponding stride-length). To define a `matrix` object, we use the aptly named function, `matrix()`.

```
mat <- matrix(data = 1:9, nrow = 3)
print(mat)

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Exercise Place the numbers `31` through `60` in a vector p. Place these same numbers in a 6×5 matrix, call it Q. *Hint:* examine your *workspace* window *after* creating these objects. How are they similar, or different?

Matrix Extraction We can access elements using the extraction operator '`[`' just as with vectors. I.e. we access element `i,j` of a matrix `mat` via `mat[i,j]`. If you wish to select all of row `i`, we simply omit the column index (e.g. `mat[i, ]`), and conversely if we wish to select an entire column.

Exercise What is the value of the 15th element in p, created in the previous exercise? Print this to console. Now, print the element in the 5th row and 3rd column of matrix Q, also created previously. Why are they the same?

Functions and Matrices Many functions also work with matrices as argument. E.g. we can calulate the sum total of elements in a matrix using `sum(mat)`.

Exercise What is the *mean* value in vector p? What about the *mean* value in matrix Q? Observe the result if we try computing other basic statistics, such as `min()`, `max()`, or `median()`.

Efficient Operations on Matrices R has many built-in functions which make matrix operations simple and performant. E.g. we can calculate column-sums or column-means via `colSums` and `colMeans` respectively.

Exercise What are the column-sums of Q? What about the column-means? Hint: see `?colSums`.

6.3 Data Frames

The canonical data structure for statistical data analysis is a `data.frame`. Unlike Python, `data.frames` are built-in to R, and unlike Matlab, columns of a `data.frame` may be referenced by name so that we don't need to remember their position.

```
df <- data.frame(y = c(11, 12, 14),
                 x = 19:21,
                 z = c(10, 0, -10))
df
```

```
##    y  x   z
## 1 11 19  10
## 2 12 20   0
## 3 14 21 -10
```

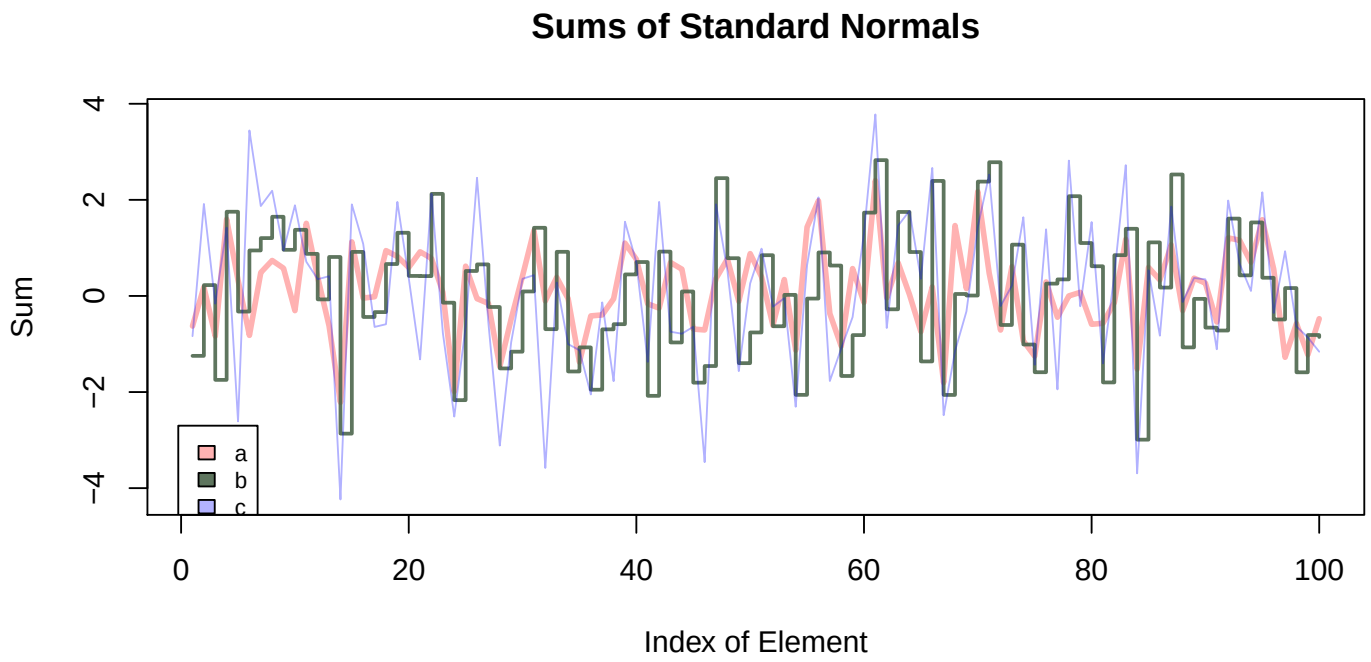
We may access into `data.frame`s using either '`$`' or '`[[`', both are extract operators, see `help("$")`.

```
mean(df$z)
mean(df[["z"]])
```

Exercise Make a script which constructs three random (standard) normal vectors of length 100. Call these vectors `x`, `y`, and `z`. Now, create a `data.frame` with columns a, b, and c which are respectively defined as x, $x+y$, and $x+y+z$. Now, plot each column separately. *Hint:* we can stack three figures vertically into a single graphics display via `par(mfrow = c(3,1))`; then, call `plot(df[["a"]])`, `plot(df[["b"]])`, and `plot(df[["c"]])`. Additionally and in a similar fashion, print out the (scalar-valued) standard deviation of each column one at a time to console (nothing to plot here). Can you explain the (statistical) result? Make sure that after your done with this exercise, you type `dev.off()` to close the graphics display that is configured to always show a triple of images in a single column.

Exercise Using the same `data.frame` created in the previous exercise, and making sure you've closed out your previous graphics device via `dev.off()`, try plotting in the following way.

```
# Set up a palette of colors. Last argument specifies opacity.
colorA <- rgb(1,   0,   0,   0.3)
colorB <- rgb(0.3, 0.4, 0.3, 0.9)
colorC <- rgb(0,   0,   1,   0.3)
# Plot our first column, then add layers to our base plot.
plot(df$a, type = "l", ylim = range(df), lwd = 3, col = colorA,
     main = "Sums of Standard Normals", xlab = "Index of Element", ylab = "Sum")
lines(df$b, type = "s", lwd = 2, col = colorB)
lines(df$c, pch  = 20, cex = 4, col = colorC)
legend(x = -0.25, y = -2.7, legend = c("a", "b", "c"), fill = c(colorA, colorB, colorC), cex = 0.75)
```



If you're curious how to interpret the graphical parameters, see `?plot` and for further detail, `?par`.

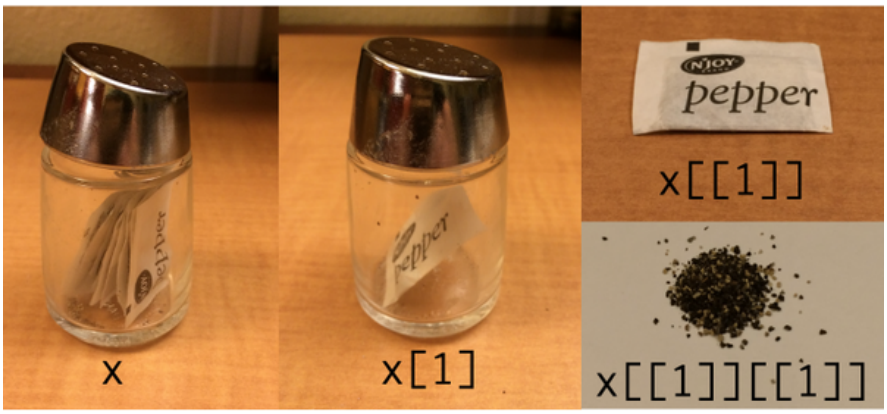
6.4 Lists

Lists are *the* fundamental data-structure in R. I.e. a `data.frame` is simply a (named) list of vectors, each the same length. In general, the elements of a list *don't* have to be the same length. And unlike matrices, the elements don't have to be of the same *type* either.

```
l <- list(one = 1, two = 1:2, five = seq(0, 1, length.out = 5))
l

## $one
## [1] 1
##
## $two
## [1] 1 2
##
## $five
## [1] 0.00 0.25 0.50 0.75 1.00

names(l)

## [1] "one"  "two"  "five"

l$five + 10

## [1] 10.00 10.25 10.50 10.75 11.00
```

Indexing into (possibly nested) lists can be difficult for beginner R users. The following satirical post can in fact be helpful.

```
# Create some salt and pepper packages.
# (Each contains a different number of particles and ratio of salt-pepper)
sap <- list(package_A = list(serv_one = rbinom(20, 1, prob = 50/100),
                             serv_two = rbinom(14, 1, prob = 30/100)),
            package_B = rbinom(28, 1, prob = 55/100),
            package_C = rbinom(36, 1, prob = 40/100))
```



Exercise Discuss the (non-rigorous) relationship between the salt-and-pepper graphic and the list `sap` (created above) with your neighbor. What's the difference between `x[1]` and `x[[1]]` in the graphic (and correspondingly in our `sap` object)?

More on List Extraction We emphasize that using the '`[[`' extract operator for lists, there are two methods of access: by name and by position. I.e. `sap[["package_C"]]` is equivalent to `sap[[3]]` since we happened to place `package_C` in the third element of our list `sap`.

7 Control Flow

This next section will be particularly useful to those entirely new to programming.

7.1 Functions

We can define our own functions in R using the following syntax:

```
Greeting <- function(name, salutation = "Hello", punctuation = "!") {
    paste(salutation, name, punctuation)
}
```

The function `Greeting` requires a `name` argument, but has default arguments set for the leading `salutation` and the trailing `punctuation` mark. We can override these by supplying the arguments explicitly.

```
Greeting("Andreas",  salutation = "god morgen")
Greeting("Santucci", salutation = "buon giorno")

## [1] "god morgen Andreas !"
## [1] "buon giorno Santucci !"
```

It's always the case that the *last expression* we use in the definition of a function acts as the return or output. We can also specify explicitly the return value. E.g. we know that we can define R-Squared for non-linear models by looking at the square of the correlation coefficient between the response y and fitted values $\hat{y}$.

```
RSQU <- function(y, x)    cor(y, x)^2                                  # Univariate case.
RSQM <- function(y, yhat) 1 - sum((y - yhat)^2) / sum((y - mean(y))^2)  # OLS: 1 - RSS / TSS.
RSQG <- function(y, yhat) cor(y, yhat)^2                               # General model.
```

Don't worry too much if you haven't seen some of these definitions before; we hope you perhaps one may be familiar.

7.2 Comparisons

The expression `a > b` for two comparable objects will return a `TRUE` if `a` is in fact strictly larger than `b`, and `FALSE` otherwise.

Exercise Write a function called `CheckAgainstThreshold(x, K=1)` which, given an input x and a threshold K (defaulting to unit value), return the result of $x > K$. Try calling it with inputs of zero, one, and two individually (keeping the threshold at $K = 1$). Additionally, pass a vector of inputs `c(0, 1, 2)`; can you explain the result to your neighbor? *Hint:* vectors and recycling.

Exercise* Write a function called `CheckID` which accepts a birthdate as argument and returns a Boolean indicating whether the corresponding person is at least K years of age, for K defaulting to 18. *Hints:* (i) use `as.Date()` to cast a string to a `Date` object, and (ii) using the function `Sys.date()` may be helpful, (iii) try for yourself subtracting two dates.

```
CheckID <- function(birthdate, K = 18) {
    # Your job is to fill in a sub-routine here.
    # Return true only if birthdate at least K years in paste.
}
```

7.3 Indexing via Booleans

We've seen previously that we can index into an object (vector, matrix, list) via individual indices. E.g. `vec[i]` or `mat[i,j]`. This will work for any i, j that are in-bounds and of the same length; we exract a sub-matrix of dimension `length(i)` $\times$ `length(j)`. We've also seen in the previous section that the `>` operator can be used to compare a vector against a single value, returning a vector of boolean values as a result.

We can also subset into a vector or matrix using logical conditions. E.g.

```
x <- runif(30)    # Sample from [0,1] uniformly at random.
idx <- x < 1/2    # In expectation, 1/2 of these evaluate TRUE.
x[idx]            # Return elements of 'x' for which 'idx' is TRUE.

##  [1] 0.473503098 0.073246870 0.423558418 0.470128618 0.120282256 0.438157397 0.431455980 0.
##  [9] 0.146561843 0.422595158 0.004766445 0.262537159 0.333605515
```

You also may find the function `which()` to be helpful when working with boolean vectors: it returns the indices of the elements whose values are `true`. Lastly, we mention that the logical negation operator `!` can invert boolean relations.

7.4 Boolean Arithmetic

```
sum(idx)
mean(idx)

## [1] 13
## [1] 0.4333333
```

You may find it interesting that we can perform arithmetic on booleans. I.e. `sum(idx)` and `mean(idx)` yield the count and proportion of values which satisfy the predicate. I.e. we count `FALSE` as zero and `TRUE` as unit valued.

Applications (Endless) The daily utility of boolean arithmetic is indispensable. I.e. want to know the proportion of GPA's above 3.5, segmented by under-graduate major? Simply create a boolean vector comparing GPA to 3.5. Apply `mean()` as an argument to `aggregate()`, i.e.

```r
d <- data.frame(major = sample(c("bio", "chem", "math", "english", "statistics"),
                               size = 100, replace = TRUE),
                gpa   = runif(100, 2.5, 4))
aggregate(d$gpa > 3.5, by = d[1], FUN = mean)
```

```
##        major         x
## 1        bio 0.4736842
## 2       chem 0.3333333
## 3    english 0.3888889
## 4       math 0.2222222
## 5 statistics 0.4285714
```

7.5 Subsetting `data.frames`

Suppose we have a `data.frame` as follows.

```r
d <- data.frame(i = letters,
                x = runif(26),
                y = rnorm(26),
                z = rnorm(26))
head(d, 3)  # See the first three...
```

```
##   i         x           y          z
## 1 a 0.5019486  0.23349984 0.01129269
## 2 b 0.4283470  1.19395526 0.99160104
## 3 c 0.6100160 -0.02790997 1.59396745
```

```r
tail(d, 1)  # ... and also last row.
```

```
##    i        x         y         z
## 26 z 0.874283 -1.183242 0.6813999
```

```r
# Pull out the first, third, and last row by ID.
idx <- d$i %in% c("a", "c", "z")
d[idx, ]
```

```
##    i         x           y          z
## 1  a 0.5019486  0.23349984 0.01129269
## 3  c 0.6100160 -0.02790997 1.59396745
## 26 z 0.8742830 -1.18324224 0.68139992
```

We could also do things like, perform conditional correlation between two columns; i.e. for observations satisfying the boolean condition (in this case, if the ID is one of "a", "c", or "z") we take a correlation.

```r
cor(d$y[idx], d$z[idx])
```

```
## [1] -0.08592514
```

8 Grouping

It's common to have data which we wish to *group* as either a pre-processing step before modeling or perhaps as post-processing in order to visualize our results. The function `table()` is used to tabulate discrete variables, while `cut()` can be used to discretize continuous variables (a.k.a. "binning"), and finally we introduce `aggregate()`.

8.1 `Table()`ing Categorical Variables

If we have a categorical variable, we can tabulate the frequency via `table()`. E.g., let's sample a thousand letters uniformly at random from the lower-case alphabet, and count the resulting labels.

```
rnd <- sample(letters, size = 1e3L, replace = TRUE)
table(rnd)

## rnd
##  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
## 49 44 47 45 46 31 39 32 37 40 48 33 35 32 34 30 33 37 31 40 43 45 44 30 33 42
```

We could also send the output of the above to `prop.table()`, which would generate a table of proportions; this is useful here since we did a simulation where its size was arbitrary, whence it makes sense to normalize the resulting frequencies by the sample size to get a measure of density.

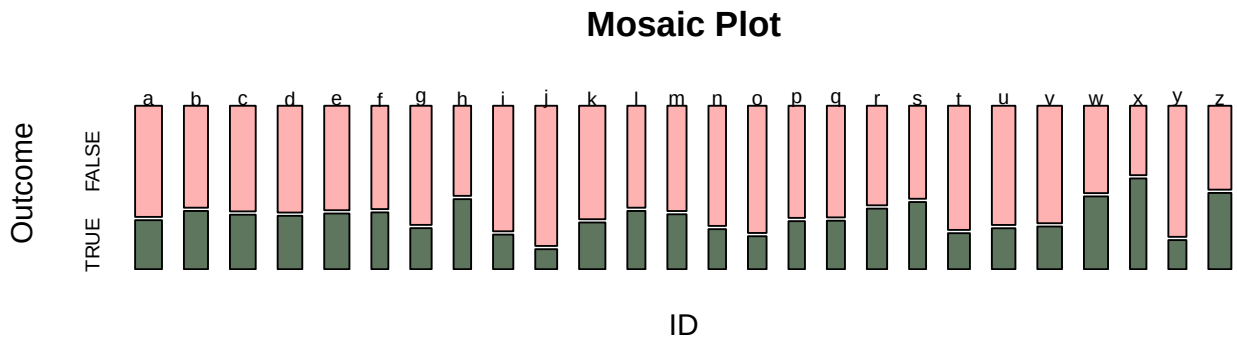
Exercise Try sending the output of the above `table()` command to a call to `plot()`. *Hint:* Since R is object-oriented, the generic `plot` method inspects the input object, recognizes it as tabulations and creates a bar-plot accordingly. See `methods("plot")` and correspondingly `?plot.table`, which is a different set of documentation from generic `?plot`.

***k*-way `tables`** The `table()` function accepts a variadic number of arguments, i.e. a *variadic function*. So, for example, if we have the randomly sampled letters above as identifiers in our data, we can table according to the results of another computation as well. E.g.

```
two_way <- table(rnd, rexp(1e3L) < 1/exp(1))
head(two_way, 3L)

##
## rnd FALSE TRUE
##   a    34   15
##   b    28   16
##   c    31   16
```

Exercise Try sending the `two_way` object we created above to a call to `plot()`. This time, we realize a Mosaic plot, which is useful for visualizing two-dimensional contingency tables.

```
plot(two_way, main = "Mosaic Plot", xlab = "ID", ylab = "Outcome", col = c(colorA, colorB))
```



8.2 Binning Continuous Variables by Quantiles

Another common application is to bin a continuous variable into buckets which have equal (empirical) density of support. E.g. maybe we want to look at deciles (or percentiles) or economic expenditures in a sector. We can use `quantile(x, probs = seq(0, 1, by = granularity))` to return a sequence of empirical quantiles. Then, we may feed these as a break-points argument when calling `cut()`. The application is as follows.

```
x <- rexp(100)
q <- quantile(x)
b <- cut(x, breaks = q, include.lowest = TRUE)
```

We now have equal support in each bucket.

```
table(b)

## b
## [0.00251,0.225]   (0.225,0.749]    (0.749,1.33]     (1.33,6.43]
##              25              25              25              25
```

8.3 Calibration Plots

Suppose we have a binomial classifier f which, given an input vector $x \in \mathbb{R}^d$ returns a fitted probability $\hat{y} \in [0, 1]$ describing the predicted probability of a positive outcome for the observation. Ideally, if our classifier is well-calibrated, our neighborhoods of predictions around a certain fitted probability $\hat{y} \pm \epsilon$, it should be the case that the actual outcome materializes in approximately $\hat{y}$ proportion of the time.

We can empirically calculate a calibration curve in R as follows, taking this moment to also demonstrate Roxygen syntax for generating documentation alongside packages, see paragraph on reproducibility and pipelines in section 10.

```
#' Given actual outcomes and fitted probabilities, estimate calibration.
#'
#' @param y A boolean vector of true outcomes.
#' @param yhat A numeric vector, the same length as input argument 'y',
#' which describes fitted probabilities (constrained to lie in [0,1]).
```

```r
#' @param granularity A scalar numeric in (0,1) interval describing
#' the granularity at which to bin our fitted values and estimate
#' efficacy of our classifier. If we choose to fine a granularity, the
#' density of support behind each point estimate will be lower.
#' @export
#' @examples
#' Calibration(0:1, rep(1/2, 2))
#' Calibration(c(rep(0, 5), rep(1, 5)), seq(0, 1, length.out = 10))
Calibration <- function(y, yhat, granularity = 1/10) {
    # We first discretize our fitted probabilities, cutting them up according to granularity
    disc <- cut(yhat, breaks = seq(from = 0, to = 1, by = granularity), include.lowest = TRUE)
    # For each discretized bin, we take the average of our response variable.
    aggregate(y, by = list(bucket = disc), FUN = mean)
}
```

A more sophisticated plotting utility may calculate no only the sample average within each `bucket` (or neighborhood of fitted probabilities) but also the standard deviation of our estimator such that we may plot a confidence interval around our point estimate for probability of detection.

9 Putting It All Together

Let's take a second to go over a simple statistical analysis, to put what we've learned together and see an R analysis wholistically.

9.1 Data Collection, Ingestion

It starts with data collection. Here, we show how you can actually use a `download.file()` function to fetch a `.zip` file from the web, then programatically `unzip()` it.

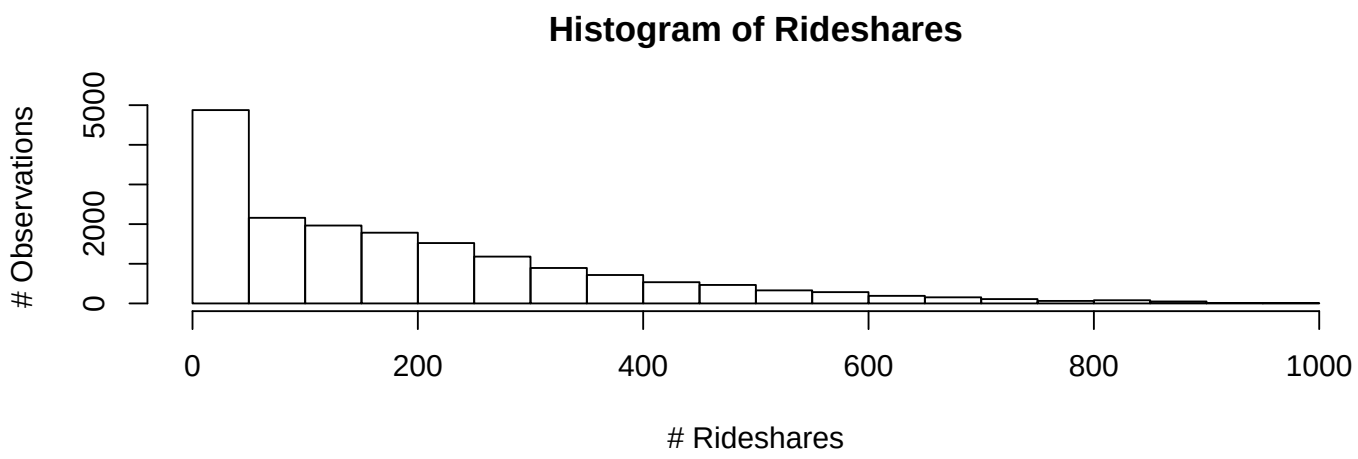
```r
# Download a zip file of bike-share data.
download.file(paste0("http://archive.ics.uci.edu/ml/machine-learning-databases/00275/",
                     "Bike-Sharing-Dataset.zip"),
              destfile = "bikesharing.zip")
# Unzip the contents and create a corresponding data directory. Load, and inspect.
unzip(zipfile = "bikesharing.zip", exdir = "bikesharing")
bshare <- read.csv("bikesharing/hour.csv")
head(bshare, 3)
```

```
##   instant     dteday season yr mnth hr holiday weekday workingday weathersit temp  atemp  hum
## 1       1 2011-01-01      1  0    1  0       0       6          0          1 0.24 0.2879 0.81
## 2       2 2011-01-01      1  0    1  1       0       6          0          1 0.22 0.2727 0.80
## 3       3 2011-01-01      1  0    1  2       0       6          0          1 0.22 0.2727 0.80
##   windspeed casual registered cnt
## 1         0      3         13  16
## 2         0      8         32  40
## 3         0      5         27  32
```

9.2 Data Visualization

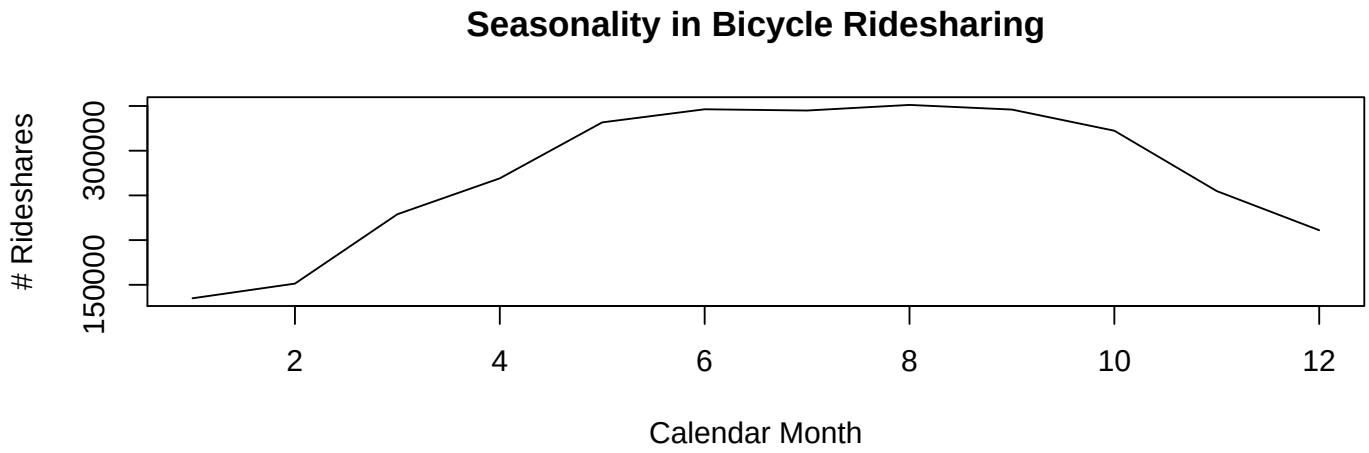
What's the univariate distribution of rideshares? See `?hist` for documentation.

```
hist(bshare$cnt, main = "Histogram of Rideshares",
     xlab = "# Rideshares", ylab = "# Observations")
```



The fact that the number of rideshares has non-trivial range which spans orders of magnitude suggests that a *log-transform* may be appropriate in a linear model. Our date-variable is already pulled apart with year, month, day, hour and metafeatures such as holiday or workday.

```
plot(aggregate(bshare$cnt, list(month = bshare$mnth), FUN = sum),
     type = "l", main = "Seasonality in Bicycle Ridesharing",
     xlab = "Calendar Month", ylab = "# Rideshares")
```



9.3 A First Model

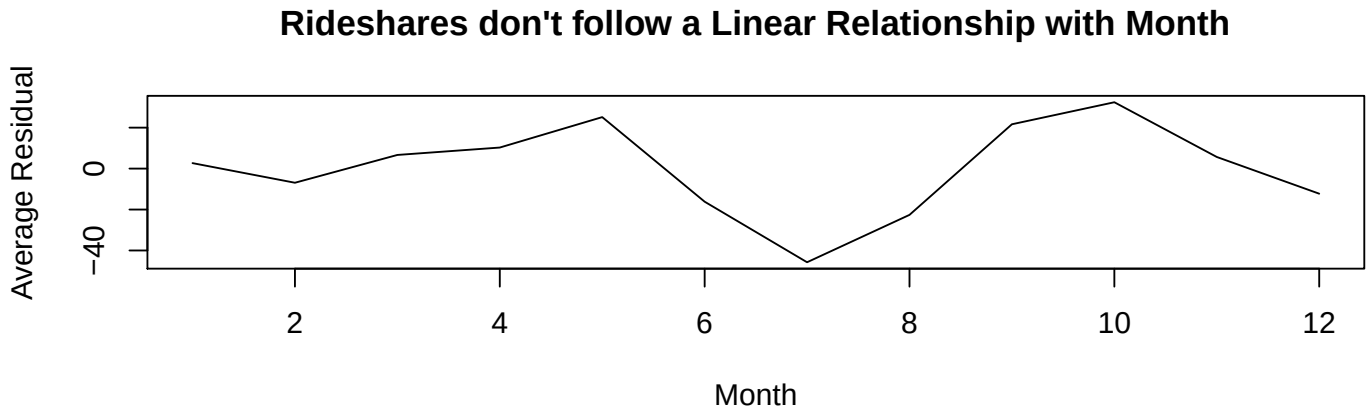
We get excited, seeing that there is a relationship between rideshares and calendar month. Let's try building a haphazard model!

```
m <- lm(cnt ~ mnth + hr + holiday + weekday + workingday + temp + hum + windspeed,
        data = bshare)
```

9.4 Checking Linear Model Assumptions

Garbage-in, garbage-out, however: our linear model assumptions were violated.

```r
# Determine if any non-linear relationships were left out.
plot(aggregate(resid(m), by = list(month = bshare$mnth), FUN = mean), type = "l",
     main = "Rideshares don't follow a Linear Relationship with Month",
     xlab = "Month", ylab = "Average Residual")
```



Yikes! We totally ignored seasonality, even though it looked like our `month` variable was statistically significant. The problem lies within the formatting of our data: the `month` variable was left as `integer`, wherein we assume that for example there is a linear relationship in ridesharing with respect to calendar month, which doesn't really make sense. Let's try again with a *factor* coding. See `?factor`, and `?I`.

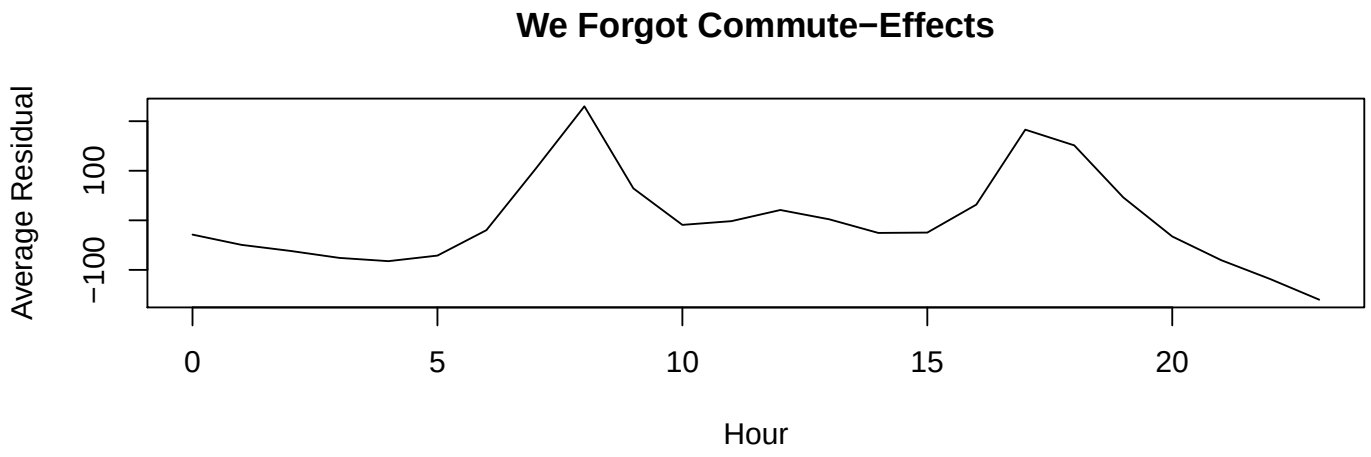
```r
m <- lm(cnt ~ I(factor(mnth)) + hr + holiday + weekday + workingday + temp + hum + windspeed,
        data = bshare)

# Verify non-linear relationships have been explicitly modeled.
all(aggregate(resid(m), by = list(month = bshare$mnth), FUN = mean)$x < 1e-7)

## [1] TRUE
```

Ah, there we go; much better! Here, we've encoded an indicator variable for each month, leaving out the first level by default (January in this case).

9.5 Iterative Model Refinement

What about our `hour` variable? It's also significant. But the same problem was made! There isn't expected to be a linear relation between hour of day and number of bicycle rideshares. There's more than likely different fixed effects, with a possibly sinusoidal pattern as a function of daylight. Easiest is to simply encode fixed effects for each hour via a factor, as before. In fact, we can use a similar code to verify the same is true for `weekday`, so we take care of that as well.

```r
# Very that our linearity assumption was violated; we see clear commute and day-of-week effe
plot(aggregate(resid(m), by = list(hour = bshare$hr),      FUN = mean), type = "l",
     main = "We Forgot Commute-Effects",    xlab = "Hour",    ylab = "Average Residual")
```



We've clearly left unexplained variation in the response on the table. Replacing factors for relevant integer-coded variables, including `workingday`.

```
m <- lm(cnt ~ I(factor(mnth)) + I(factor(hr)) + I(factor(weekday)) + I(factor(workingday)) + h
           temp + hum + windspeed,
         data = bshare)
# summary(m)
```

And of course, here we realize that since we one-hot encoded each day-of-week, our binary indicator for `workingday` becomes colinear, whence non-estimable in our OLS. R returns an NA coefficient for one of the colinear variables as expected.

9.6 Binning Continuous Variables

But what about `windspeed`? This is certainly a continuous variable, and so different from our previous integer variables we've modified into factors previously. But do we expect windspeed to have a linear relationship with `rideshare`s? In fact this assumption is strongly violated.

```
res <- aggregate(resid(m), by = list(bshare$windspeed), FUN = mean)
res[seq(1, nrow(res), length.out = 6), ]

##    Group.1           x
## 1   0.0000   -9.166958
## 6   0.1940    3.169384
## 12  0.3582   -5.679233
## 18  0.5224   -5.355442
## 24  0.6866   48.846713
## 30  0.8507 -160.264846
```

Here, we need to update our regression to instead use a *binned* analogue of wind-speed. We decide on using quantiles.

```
bshare$wind_grp <- cut(bshare$windspeed, quantile(bshare$windspeed), include.lowest = TRUE)
m <- lm(cnt ~ I(factor(mnth)) + I(factor(hr)) + I(factor(weekday)) +
           holiday + workingday + temp + hum + wind_grp,
         data = bshare)
# summary(m)
```

There's many further refinements we could likely make, but we've given a pattern for fine-tuning our statistical models via diagnostic models and graphical summaries. We can even try calling `plot()` on our linear model object to inspect further diagnostics.

Exercise We used `summary()` once on a `data.frame` object, whereby we received numerical summary statistics for each column, and another time on a linear model object, in which case we were given inference statistics for our regression output. What other summaries are possible in base R? *Hint:* see `methods("summary")` and then correspondingly `?summary.<method_of_interest>`.

10 What Next?

This course is a simple introduction, and we hope to have sparked an appetite for learning more about statistical computing (in R)! We've include some resources in the bibliography below.

Nailing the Basics I found that R in a Nutshell was a great thorough reference.

Packages in R So far we've stuck with base R. To install a package for the first time (after becoming aware of its existence through a web search or word of mouth), we may type `install.packages("<package_name")` to download and install the binary files required to call upon the package contents. We then must type `library("<package_name")` at the start of each R session in which we wish to use said package. If you wish to see a listing of available objects that come exported with a package, try `help(package = "<package_name>")`; from there, you can inquire for further help on individual function `help(package_name::function_name)`.

Tidyverse Recently popular in the R programming community, the tidyverse introduction to R (click the table-of-contents in the upper right corner to access full course contents) by Hadley Wickham seeks to rewrite base R in a more user friendly way to further enable interactive statistical programming. The packages rely on the `magrittr` package to pipe computations forward in a more readable manner. From there, I recommend learning from Hadley's excellent resource on Advanced R programming.

Visualization We used base R for our graphics, which can go a long way. The package `ggplot2`, also by Hadley, provides a grammar of graphics approach to data visualization. From there, you can easily learn animations via `gganimate`, or even plotting geospatial (mapping) data via `ggmap`.

Performant Computing In a completely different direction, one of my favorite packages in R is `data.table`, authored by Matt Dowle. It supports assignment by reference, and allows us to compute on larger data sets more efficiently. Advantages we can already see based on examples above: each call to `aggregate()` can be replaced by a `group-by` operation within the `data.table`, where we can take advantage of scoping to forego typing `dt$` constantly to select variables. If you're interested in performant computing in R, CRAN has an entire page dedicated to this subject, pointing to libraries geared toward computing with large data efficiently.

Pipelines / Reproducibility If you're writing a collection of R scripts that are all related, you may consider bundling them into your own R package. You can then add unit-tests easily via `testthat`, and have these run automatically each time you rebuild your package. If you want to be really fancy, you can integrate these unit tests in with your Git workflow via Continuous Integration, where we emphasize the aspect of having the tests re-run with each new commit (e.g. Travis).

11 Exercises

Exercises marked with `(*)` may be slightly more difficult.

1. You are given an input of numeric values x, and are told that instead of corrupted observations being reported as missing, they are instead reported as the numeric value `999`. E.g. a sample input could be `x = c(1, 4, 2, 999, 7, 11, 999, 999, 12)`. Use `ifelse` to create a new object which replaces any instance of `999` with `NA`, which is R's internal representation for "not available" or missing data. Then, use `sum()`, `!`, and `is.na()` to count the number of non-missing entries provided in the input.

2. You are given an input of consecutive integers between 1 and N inclusive, except exactly *one element* is missing. Your goal is to write a function which returns the missing element. E.g. `FindMissing(c(1,2,4,5))` should return `3`. *Hints:* try solving several ways: (i) by using operators `which()`, `%in%`, and `!`, (ii) by using `setdiff()`, `range()`, and `seq()`, and (iii) using triangular numbers.

3. Make a vector of consecutive integers from 1 to 100 inclusive. Write a `for`-loop which runs through the whole vector, multiplying elements less than 5 or larger than 90 by 10 and all other elements by 1/10. Then, rewrite your above solution using what you learn from examining `?ifelse`.

4. You are given an input sequence x and an output sequence $f(x)$. Write a function which returns the value of x corresponding to the maximum of $f(x)$. I.e. implement an `argmax` function that is of zeroth-order. *Hint:* see `?which.max`, and use the result in conjuction with an `'['` operator.

5. You are given a single string with a listing of words. Return a table describing the frequency of word-occurrences. *Hint:* you'll find the functions `strsplit()`, `unlist()`, and `table()`, helpful.

6. You are given a folder (working directory) with consumption data stored in `.csv` files which you need to load into memory. There are 12 files, one for each calendar month of a particular year. Your goal is to use `list.files()` and a `for()` loop to iteratively `rbind()` the `data.frame`'s together. You'll need to take care to add a month describing the calendar month during the process, since the raw data won't include this. To get started, run the following code to create 12 sample data files in a new directory.

```
dir.create("calendar_months_data")
for (i in 1:12) {
    d <- data.frame(id = letters, y  = rnorm(26))
    write.csv(d, file = paste0("calendar_months_data/month_", i, ".csv"),
              row.names = FALSE)
}
```

   Once you solve the problem iteratively using `rbind()`, try to understand what the following expression does. `l <- lapply(fnames, read.csv); do.call(rbind, l)`. See `?sapply()` which replaces a `for`-loop, and `do.call()`.

7. (*) Rewrite our data generation process used in the above question as follows: first write a function `Create(i)` which takes an integer i and effectively replaces the body of our `for`-loop above. Then, realize that we can simply write `for (i in 1:12) Create(i)` to achieve the same result! Now, look at `?sapply` and figure out how to use it to achieve again the identical result.

8. (*) You are again given a single string with a listing of words. Return a frequency count of all n-grams occurring, for arbitrary n. We define an n-gram as a contiguous sequence of n words. *Hint:* start with $n = 2$.

9. (*) Given an input stream 1,000 digits long, find the greatest product of five consecutive digits. (The input is given as a single string. Your output should be a single scalar.)

10. (*) Given an integer K, return a description of the number of days in each of the twelve calendar months; some years are leap years, in which case February will have 29 days. *Hint:* see `?seq.Date`.

11. (*) You are given an urn with an unlimited number of balls in it, where you are told that 4/10 of them are red, 1/2 are green, and the remaining 1/10 are blue. You draw a sequence of balls until you draw one which is blue. How many draws do you expect to take before realizing a success? *Hint:* you can solve this problem using built-in statistical functions if you understand the probability distribution we are drawing from, and alternatively you can also solve it via simulation. You may find `?replicate` to be helpful if you choose the latter approach.

12. (*) What's more likely, the sequence of coin tosses `HTH` or `HHT`? Support your answer with a simulation in R. This answer can also be answered using theory from Markov chains.

13. (*) What's the difference between `sapply()` and `lapply()` exactly? Can you make one behave like the other? *Hint:* see `head(sapply)` and also `?sapply`.

14. (*) Why would you use `mapply()`? As a trivial example, consider re-implementing element-wise vector addition in terms of a single (and simple) call to `mapply` where we supply `FUN = "+"`. *Hint:* see `help("+")` to learn the names of the arguments the addition operator expects.

References

[1] Adler, Joseph. R in a Nutshell 2012, O'Reilly

[2] Brauer, Torfs. *A (very) short introduction to R*. 2014, CRAN.

[3] Burns, Patrick. The R Inferno 2011

[4] Free resources for learning R 2016, Stack Exchange

[5] Abelson and Susman Structure and Interpretation of Computer Programs MIT, 2005

[6] An Introduction to R CRAN, July 2018

[7] Dirk Eddelbuettel CRAN Task View: High-Performance & Parallel Computing with R July 30, 2018

[8] Hilary Parker Writing an R Package from scratch April 29, 2014

[9] Jonathan Taylor Data Science 101 Spring 2018