Lab 0.1: Introduction to R

Introduction to Statistical Computing

Luis Torres Serrano

13/01/2025

Contents

This lab is to be done outside of class time. You may collaborate with one classmate, but you must identify yourself and his/her name above, in the author's field, and you must submit **your own** lab as this completed .Rmd file.

```
## For reproducibility --- don't change this!
set.seed(150)
```

Hint #1. The binomial distribution

The binomial distribution $\text{Bin}(m, p)$ is defined by the number of successes in m independent trials, each have probability p of success. Think of flipping a coin m times, where the coin is weighted to have probability p of landing on heads.

The R function `rbinom()` generates random variables with a binomial distribution. E.g.,

```
rbinom(n=20, size=10, prob=0.5)
```

produces 20 observations from $\text{Bin}(10, 0.5)$.

Q1. Some simple manipulations

1a. Generate 500 random values from the $\text{Bin}(15, 0.5)$ distribution, and store them in a vector called `bin.draws.0.5`. Extract and display the first 25 elements. Extract and display all but the first 475 elements.

```
# YOUR CODE GOES HERE
```

1b. Try the different methods for selecting the elements of the vector.

```
# YOUR CODE GOES HERE
```

1c. Use function `length` to get the size of the `bin.draws.0.5` vector

```
# YOUR CODE GOES HERE
```

1d. Add the first element of `bin.draws.0.5` to the fifth. Compare the second element to the tenth, which is larger? A bit more tricky: print the indices of the elements of `bin.draws.0.5` that are equal to 3. How many such elements are there? Theoretically, how many such elements would you expect there to be? Hint: it would be helpful to look at the help file for the `rbinom()` function.

```
# YOUR CODE GOES HERE
```

1e. Find the mean and standard deviation of `bin.draws.0.5`. Is the mean close what you'd expect? The standard deviation?

```
# YOUR CODE GOES HERE
```

1f. Call `summary()` on `bin.draws.0.5` and describe the result.

```
# YOUR CODE GOES HERE
```

1g. Call `str()` on `bin.draws.0.5` and describe the result.

```
# YOUR CODE GOES HERE
```

1h. Find the data type of the elements in `bin.draws.0.5` using `typeof()`. Then convert `bin.draws.0.5` to a vector of characters, storing the result as `bin.draws.0.5.char`, and use `typeof()` again to verify that you've done the conversion correctly. Call `summary()` on `bin.draws.0.5.char`. Is the result formatted differently from what you saw above? Why?

```
# YOUR CODE GOES HERE
```

Q2. Some simple plots

2a. The function `plot()` is a generic function in R for the visual display of data. The function `hist()` specifically produces a histogram display. Use `hist()` to produce a histogram of your random draws from the binomial distribution, stored in `bin.draws.0.5`.

```
# YOUR CODE GOES HERE
```

2b. Call `tabulate()` on `bin.draws.0.5`. What is being shown? Does it roughly match the histogram you produced in the last question?

```
# YOUR CODE GOES HERE
```

2c. Call `plot()` on `bin.draws.0.5` to display your random values from the binomial distribution. Can you interpret what the `plot()` function is doing here?

```
# YOUR CODE GOES HERE
```

2d. Call `plot()` with two arguments, the first being `1:500`, and the second being `bin.draws.0.5`. This creates a scatterplot of `bin.draws.0.5` (on the y-axis) versus the indices 1 through 500 (on the x-axis). Does this match your plot from the last question?

```
# YOUR CODE GOES HERE
```

Q3. More binomials, more plots

3a. Generate 500 binomials again, composed of 15 trials each, but change the probability of success to: 0.2, 0.3, 0.4, 0.6, 0.7, and 0.8, storing the results in vectors called `bin.draws.0.2`, `bin.draws.0.3`, `bin.draws.0.4.`, `bin.draws.0.6`, `bin.draws.0.7` and `bin.draws.0.8`. For each, compute the mean and standard deviation.

```
# YOUR CODE GOES HERE
```

3b. We'd like to compare the properties of our vectors. Create a vector of length 7, whose entries are the means of the 7 vectors we've created, in order according to the success probabilities of their underlying binomial distributions (0.2 through 0.8).

```
# YOUR CODE GOES HERE
```

3c. Use the `names` function to change the name of the vector samples by `c("bin.draws.0.2", "bin.draws.0.3", "bin.draws.0.4", "bin.draws.0.6", "bin.draws.0.7", "bin.draws.0.8")`.

```
# YOUR CODE GOES HERE
```

3d. Using the vectors from the last part, create the following scatterplots. Explain in words, for each, what's going on. * The 7 means versus the 7 probabilities used to generate the draws. * The standard deviations versus the probabilities. * The standard deviations versus the means.

Challenge: for each plot, add a curve that corresponds to the relationships you'd expect to see in the theoretical population (i.e., with an infinite amount of draws, rather than just 500 draws).

```
# YOUR CODE GOES HERE
```

Q4. Working with matrices

4a. Create a matrix of dimension 500 x 7, called `bin.matrix`, whose columns contain the 7 vectors we've created, in order of the success probabilities of their underlying binomial distributions (0.2 through 0.8). Hint: use `cbind()`.

```
# YOUR CODE GOES HERE
```

4b. Print the first five rows of `bin.matrix`. Print the element in the 66th row and 5th column. Compute the largest element in first column. Compute the largest element in all but the first column.

```
# YOUR CODE GOES HERE
```

4c. Calculate the column means of `bin.matrix` by using just a single function call.

```
# YOUR CODE GOES HERE
```

4d. Compare the means you computed in the last question to those you computed in Q3b, in two ways. First, using `==`, and second, using `identical()`. What do the two ways report? Are the results compatible? Explain.

```
# YOUR CODE GOES HERE
```

4e. Take the transpose of `bin.matrix` and then take row means. Are these the same as what you just computed? Should they be?

```
# YOUR CODE GOES HERE
```

Q5. Warm up is over, let's go big

5a. R's capacity for data storage and computation is very large compared to what was available 10 years ago. Generate 5 million numbers from $\text{Bin}(1 \times 10^6, 0.5)$ distribution and store them in a vector called `big.bin.draws`. Calculate the mean and standard deviation of this vector.

```
# YOUR CODE GOES HERE
```

5b. Create a new vector, called `big.bin.draws.standardized`, which is given by taking `big.bin.draws`, subtracting off its mean, and then dividing by its standard deviation. Calculate the mean and standard deviation of `big.bin.draws.standardized`. (These should be 0 and 1, respectively, or very close to it; if not, you've made a mistake somewhere).

```
# YOUR CODE GOES HERE
```

5d. Calculate the proportion of times that an element of `big.bin.draws.standardized` exceeds 1.644854. Is this close to 0.05?

```
# YOUR CODE GOES HERE
```

Q6. Now let's go really big

6a. Let's push R's computational engine a little harder. Generate 200 million numbers from Bin($10 \times 10^6, 50 \times 10^{-8}$), and save it in a vector called `huge.bin.draws`.

```
# YOUR CODE GOES HERE
```

6b. Calculate the mean and standard deviation of `huge.bin.draws`. Are they close to what you'd expect? (They should be very close.) Did it longer to compute these, or to generate `huge.bin.draws` in the first place?

```
# YOUR CODE GOES HERE
```

6c. Calculate the median of `huge.bin.draws`. Did this median calculation take longer than the calculating the mean? Is this surprising?

```
# YOUR CODE GOES HERE
```

6d. Calculate the exponential of the median of the logs of `huge.bin.draws`, in one line of code. Did this take longer than the median calculation applied to `huge.bin.draws` directly? Is this surprising?

```
# YOUR CODE GOES HERE
```

6e. Plot a histogram of of `huge.bin.draws`, again with a large setting of the `breaks` argument (e.g., `breaks=100`). Describe what you see; is this different from before, when we had 3 million draws? **Challenge**: Is this surprising? What distribution is this?

```
# YOUR CODE GOES HERE
```

Q7. Going big with lists

7a. Convert `big.bin.draws` into a list using `as.list()` and save the result as `big.bin.draws.list`. Check that you indeed have a list by calling `class()` on the result. Check also that your list has the right length, and that its 1159th element is equal to that of `big.bin.draws`.

```
# YOUR CODE GOES HERE
```

7b. Run the code below, to standardize the binomial draws in the list `big.bin.draws.list`. Note that `lapply()` applies the function supplied in the second argument to every element of the list supplied in the first argument, and then returns a list of the function outputs. (We'll learn much more about the `apply()` family of functions later in the course.) Did this `lapply()` command take longer to evaluate than the code you wrote in Q5b? (It should have; otherwise your previous code could have been improved, so go back and improve it.) Why do you think this is the case?

```
big.bin.draws.mean = mean(big.bin.draws)
big.bin.draws.sd = sd(big.bin.draws)
standardize = function(x) {
  return((x - big.bin.draws.mean) / big.bin.draws.sd)
}
big.bin.draws.list.standardized.slow = lapply(big.bin.draws.list, standardize)
```

7c. Run the code below, which again standardizes the binomial draws in the list `big.bin.draws.list`, using `lapply()`. Why is it so much slower than the code in the last question? (You may stop evaluation if it is taking too long!) Think about what is happening each time the function is called.

```
standardize.slow = function(x) {
  return((x - mean(big.bin.draws)) / sd(big.bin.draws))
}
big.bin.draws.list.standardized.slow = lapply(big.bin.draws.list, standardize.slow)
```

7d. Lastly, let's look at memory usage. The command `object.size(x)` returns the number of bytes used to store the object `x` in your current R session. Find the number of bytes used to store `big.bin.draws` and `big.bin.draws.list`. How many megabytes (MB) is this, for each object? Which object requires more memory, and why do you think this is the case? Remind yourself: why are lists special compared to vectors, and is this property important for the current purpose (storing the binomial draws)?

```
# YOUR CODE GOES HERE
```

Hint #2. Cleaning the workspace and freeing memory

We can check which variables we have loaded in the workspace using the `ls()` function and delete variables from the workspace using the `rm()` function. To delete all the variables in the workspace:

```
rm(list = ls())
```

However, this does not involve to free the memory that was occupied by the objects (you can use `memory.size()` to check how much memory is being used). `gc()` shall be used to call the garbage collector and return some of the memory that was lost. **YOU SHALL NOT REPEATEDLY USED THIS FUNCTION, USE IT ONLY WHEN THERE ARE NO ALTERNATIVE LIKE KEEPING TASKS IN R FILES AND EXECUTE THEM USING THE `littler` PACKAGE**

```
gc()
```

```
##           used (Mb) gc trigger (Mb) max used (Mb)
## Ncells  557930 29.8    1259205 67.3  1259205 67.3
## Vcells 1074891  8.3    8388608 64.0  2093842 16.0
```

Hint #3. Loading data from files

To read a data from a *.csv*, *.txt* or a *.dat* file, the most used option is the *read.table*() function with a suitable *sep* argument depending on the file. You can execute ?*read.table* to see the help documentation for this function with usage examples.

Q8. Let's revise ggplot2

For **Simdata1.csv** to **Simdata7.csv** and fill the *LabPractice_0_Table.docx* file:

8a Identify if the dataset corresponds to a regression or a classification problem. In the latter, identify if it is a binary or a multiclass problem. Convert the output variable to factors if needed.

```
# YOUR CODE GOES HERE
```

8b Does the data need preprocessing? Check if there are outliers or missing values. If any, how many are there?

```
# YOUR CODE GOES HERE
```

8c Use `ggplot2` for plotting a 2D scatterplot of the data. If Y is categorical, make sure to change the color or the shape of the data depending on the values of Y.

```
# YOUR CODE GOES HERE
```

8d Can you find the solution of the problem? Can you express the solution mathematically?

```
# YOUR CODE GOES HERE
```

For **Simdata7.csv**:

8e Plot the histograms of the input and output variables.

```
# YOUR CODE GOES HERE
```

8f Divide the histogram plot into subplots based on the values of the output variable using the `facet_wrap` function.

```
# YOUR CODE GOES HERE
```

8g Apply `ggpairs` function dividing the results based on the values of the output variable.

```
# YOUR CODE GOES HERE
```

Q9. More complicated basics

9a. Let's start easy by working through some R basics, to continue to brush up on them. Define a variable `x.vec` to contain the integers 1 through 100. Check that it has length 100. Report the data type being stored in `x.vec`. Add up the numbers in `x.vec`, by calling a built-in R function. How many arithmetic operations did this take? **Challenge**: show how Gauss would have done this same calculation as a 7 year old, using just 3 arithmetic operations.

```
# YOUR CODE GOES HERE
```

9b. Convert `x.vec` into a matrix with 20 rows and 5 columns, and store this as `x.mat`. Here `x.mat` should be filled out in the default order (column major order). Check the dimensions of `x.mat`, and the data type as well. Compute the sums of each of the 5 columns of `x.mat`, by calling a built-in R function. Check (using a comparison operator) that the sum of column sums of `x.mat` equals the sum of `x.vec`.

```
# YOUR CODE GOES HERE
```

9c. Extract and display rows 1, 5, and 17 of `x.mat`, with a single line of code. Answer the following questions, each with a single line of code: how many elements in row 2 of `x.mat` are larger than 40? How many elements in column 3 are in between 45 and 50? How many elements in column 5 are odd? Hint: take advantage of the `sum()` function applied to Boolean vectors.

```
# YOUR CODE GOES HERE
```

9d. Using Boolean indexing, modify `x.vec` so that every even number in this vector is incremented by 10, and every odd number is left alone. This should require just a single line of code. Print out the result to the console. **Challenge**: show that `ifelse()` can be used to do the same thing, again using just a single line of code.

```
# YOUR CODE GOES HERE
```

9e. Consider the list `x.list` created below. Complete the following tasks, each with a single line of code: extract all but the second element of `x.list`—seeking here a list as the final answer. Extract the first and third elements of `x.list`, then extract the second element of the resulting list—seeking here a vector as the final answer. Extract the second element of `x.list` as a vector, and then extract the first 10 elements of this vector—seeking here a vector as the final answer. Note: pay close attention to what is asked and use either single brackets `[ ]` or double brackets `[[ ]]` as appropriate.

```
x.list = list(rnorm(6), letters, sample(c(TRUE,FALSE),size=4,replace=TRUE))
# YOUR CODE GOES HERE
```

Prostate cancer data set

We're going to look at a data set on 97 men who have prostate cancer (from the book The Elements of Statistical Learning). There are 9 variables measured on these 97 men:

1. `lpsa`: log PSA score
2. `lcavol`: log cancer volume
3. `lweight`: log prostate weight
4. `age`: age of patient
5. `lbph`: log of the amount of benign prostatic hyperplasia
6. `svi`: seminal vesicle invasion
7. `lcp`: log of capsular penetration
8. `gleason`: Gleason score
9. `pgg45`: percent of Gleason scores 4 or 5

To load this prostate cancer data set into your R session, and store it as a matrix `pros.dat`:

```
pros.dat =
  as.matrix(read.table("http://www.stat.cmu.edu/~ryantibs/statcomp/data/pros.dat"))
```

Q10. Basic indexing and calculations

10a. What are the dimensions of `pros.dat` (i.e., how many rows and how many columns)? Using integer indexing, print the first 6 rows and all columns; again using integer indexing, print the last 6 rows and all columns.

```
# YOUR CODE GOES HERE
```

10b. Using the built-in R functions `head()` and `tail()` (i.e., do *not* use integer indexing), print the first 6 rows and all columns, and also the last 6 rows and all columns.

```
# YOUR CODE GOES HERE
```

10c. Does the matrix `pros.dat` have names assigned to its rows and columns, and if so, what are they? Use `rownames()` and `colnames()` to find out. Note: these would have been automatically created by the `read.table()` function that we used above to read the data file into our R session. To see where `read.table()` would have gotten these names from, open up the data file: http://www.stat.cmu.edu/~ryantibs/statcomp/data/pros.dat in your web browser. Only the column names here are actually informative.

```
# YOUR CODE GOES HERE
```

10d. Using named indexing, pull out the two columns of `pros.dat` that measure the log cancer volume and the log cancer weight, and store the result as a matrix `pros.dat.sub`. (Recall the explanation of variables at the top of this lab.) Check that its dimensions make sense to you, and that its first 6 rows are what you'd expect. Did R automatically assign column names to `pros.dat.sub`?

```
# YOUR CODE GOES HERE
```

10e. Using the log cancer weights and log cancer volumes, calculate the log cancer density for the 97 men in the data set (note: density = weight / volume). There are in fact two different ways to do this; the first uses three function calls and one arithmetic operation; the second just uses one arithmetic operation. Note: in either case, you should be able to perform this computation for all 97 men *with a single line of code*, taking advantage of R's ability to vectorize. Write code to do it both ways, and show that both ways lead to the same answer, using `all.equal()`.

```
# YOUR CODE GOES HERE
```

10f. Append the log cancer density to the columns of `pros.dat`, using `cbind()`. The new `pros.dat` matrix should now have 10 columns. Set the last column name to be `ldens`. Print its first 6 rows, to check that you've done all this right.

```
# YOUR CODE GOES HERE
```

Q11. Exploratory data analysis with plots

11a. Using `hist()`, produce a histogram of the log cancer volume measurements of the 97 men in the data set; also produce a histogram of the log cancer weight. In each case, use `breaks=20` as an arugment to `hist()`. Comment just briefly on the distributions you see. Then, using `plot()`, produce a scatterplot of the log cancer volume (y-axis) versus the log cancer weight (x-axis). Do you see any kind of relationship? Would you expect to? **Challenge**: how would you measure the strength of this relationship formally? Note that there is certainly more than one way to do so. We'll talk about statistical modeling tools later in the course.

```
# YOUR CODE GOES HERE
```

11b. Produce scatterplots of log cancer weight versus age, and log cancer volume versus age. Do you see relationships here between the age of a patient and the volume/weight of his cancer?

```
# YOUR CODE GOES HERE
```

11c. Produce a histogram of the log cancer density, and a scatterplot of the log cancer density versus age. Comment on any similarities/differences you see between these plots, and the corresponding ones you produced above for log cancer volume/weight.

```
# YOUR CODE GOES HERE
```

11d. Delete the last column, corresponding to the log cancer density, from the `pros.dat` matrix, using negative integer indexing.

```
# YOUR CODE GOES HERE
```

Q12. A bit of Boolean indexing never hurt anyone

12a. The `svi` variable in the `pros.dat` matrix is binary: 1 if the patient had a condition called "seminal vesicle invasion" or SVI, and 0 otherwise. SVI (which means, roughly speaking, that the cancer invaded into the muscular wall of the seminal vesicle) is bad: if it occurs, then it is believed the prognosis for the patient is poorer, and even once/if recovered, the patient is more likely to have prostate cancer return in the future. Compute a Boolean vector called `has.svi`, of length 97, that has a `TRUE` element if a row (patient) in `pros.dat` has SVI, and `FALSE` otherwise. Then using `sum()`, figure out how many patients have SVI.

```
# YOUR CODE GOES HERE
```

12b. Extract the rows of `pros.dat` that correspond to patients with SVI, and the rows that correspond to patients without it. Call the resulting matrices `pros.dat.svi` and `pros.dat.no.svi`, respectively. You can do this in two ways: using the `has.svi` Boolean vector created above, or using on-the-fly Boolean indexing, it's up to you. Check that the dimensions of `pros.dat.svi` and `pros.dat.no.svi` make sense to you.

```
# YOUR CODE GOES HERE
```

12c. Using the two matrices `pros.dat.svi` and `pros.dat.no.svi` that you created above, compute the means of each variable in our data set for patients with SVI, and for patients without it. Store the resulting means into vectors called `pros.dat.svi.avg` and `pros.dat.no.svi.avg`, respectively. Hint: for each matrix, you can compute the means with a single call to a built-in R function. What variables appear to have different means between the two groups?

```
# YOUR CODE GOES HERE
```

Q13. Computing standard deviations using iteration

13a. Take a look at the starter code below. The first line defines an empty vector `pros.dat.svi.sd` of length `ncol(pros.dat)` (of length 9). The second line defines an index variable `i` and sets it equal to 1. Write a third line of code to compute the standard deviation of the `i`th column of `pros.dat.svi`, using a built-in R function, and store this value in the `i`th element of `pros.dat.svi.sd`.

```
pros.dat.svi.sd = vector(length=ncol(pros.dat))
i = 1
# YOUR CODE GOES HERE
```

13b. Repeat the calculation as in the previous question, but for patients without SVI. That is, produce three lines of code: the first should define an empty vector `pros.dat.no.svi.sd` of length `ncol(pros.dat)` (of length 9), the second should define an index variable `i` and set it equal to 1, and the third should fill the `i`th element of `pros.dat.no.svi.sd` with the standard deviation of the `i`th column of `pros.dat.no.svi`.

```
pros.dat.no.svi.sd = vector(length=ncol(pros.dat))
i = 1
# YOUR CODE GOES HERE
```

13c. Write a `for()` loop to compute the standard deviations of the columns of `pros.dat.svi` and `pros.dat.no.svi`, and store the results in the vectors `pros.dat.svi.sd` and `pros.dat.no.svi.sd`, respectively, that were created above. Note: you should have a single `for()` loop here, not two for loops. And if it helps, consider breaking this task down into two steps: as the first step, write a `for()` loop that iterates an index variable `i` over the integers between 1 and the number of columns of `pros.dat` (don't just manually write 9 here, pull out the number of columns programmatically), with an empty body. As the second step, paste relevant pieces of your solution code from Q5a and Q5b into the body of the `for()` loop. Print out the resulting vectors `pros.dat.svi.sd` and `pros.dat.no.svi.sd` to the console. Comment, just briefly (informally), by visually inspecting these standard deviations and the means you computed in Q4c: which variables exhibit large differences in means between the SVI and non-SVI patients, relative to their standard deviations?

```
# YOUR CODE GOES HERE
```

13d. The code below computes the standard deviations of the columns of `pros.dat.svi` and `pros.dat.no.svi`, and stores them in `pros.dat.svi.sd.master` and `pros.dat.no.svi.sd.master`, respectively, using `apply()`. (We'll learn `apply()` and related functions a bit later in the course.) Remove `eval=FALSE` as an option to the Rmd code chunk, and check using `all.equal()` that the standard deviations you computed in the previous question equal these "master" copies. Note: use `check.names=FALSE` as a third argument to `all.equal()`, which instructs it to ignore the names of its first two arguments. (If `all.equal()` doesn't succeed in both cases, then you must have done something wrong in computing the standard deviations, so go back and fix them!)

```
pros.dat.svi.sd.master = apply(pros.dat.svi, 2, sd)
pros.dat.no.svi.sd.master = apply(pros.dat.no.svi, 2, sd)
all.equal(pros.dat.svi.sd, pros.dat.svi.sd.master, check.names=FALSE)
all.equal(pros.dat.no.svi.sd, pros.dat.no.svi.sd.master, check.names=FALSE)
```

Q14. Computing t-tests using vectorization

14a. Recall that the **two-sample (unpaired) t-statistic** between data sets $X = (X_1, \ldots, X_n)$ and $Y = (Y_1, \ldots, Y_m)$ is:

$$T = \frac{\bar{X} - \bar{Y}}{\sqrt{\frac{s_X^2}{n} + \frac{s_Y^2}{m}}},$$

where $\bar{X} = \sum_{i=1}^n X_i/n$ is the sample mean of X, $s_X^2 = \sum_{i=1}^n (X_i - \bar{X})^2/(n-1)$ is the sample variance of X, and similarly for $\bar{Y}$ and s_Y^2. We will compute these t-statistics for all 9 variables in our data set, where X will play the role of one of the variables for SVI patients, and Y will play the role of this variable for non-SVI patients. Start by computing a vector of the denominators of the t-statistics, called `pros.dat.denom`, according to the formula above. Take advantage of vectorization; this calculation should require just a single line of code. Make sure not to include any hard constants (e.g., don't just manually write 21 here for n); as always, programmatically define all the relevant quantities. Then compute a vector of t-statistics for the 9 variables in our data set, called `pros.dat.t.stat`, according to the formula above, and using `pros.dat.denom`. Again, take advantage of vectorization; this calculation should require just a single line of code. Print out the t-statistics to the console.

```
# YOUR CODE GOES HERE
```

14b. Given data X and Y and the t-statistic T as defined the last question, the **degrees of freedom** associated with T is:

$$\nu = \frac{(\frac{s_X^2}{n} + \frac{s_Y^2}{m})^2}{\frac{(\frac{s_X^2}{n})^2}{n-1} + \frac{(\frac{s_Y^2}{m})^2}{m-1}}.$$

Compute the degrees of freedom associated with each of our 9 t-statistics (from our 9 variables), storing the result in a vector called `pros.dat.df`. This might look like a complicated/ugly calculation, but really, it's not too bad: it only involves arithmetic operators, and taking advantage of vectorization, the calculation should only require a single line of code. Hint: to simplify this line of code, it will help to first set short variable names for variables/quantities you will be using, as in `sx = pros.dat.svi.sd`, `n = nrow(pros.dat.svi)`, and so on. Print out these degrees of freedom values to the console.

```
# YOUR CODE GOES HERE
```

14c. The function `pt()` evaluates the distribution function of the t-distribution (example below) and returns the probability that a t-distributed random variable, with `v` degrees of freedom, exceeds the value `x`. Importantly, `pt()` is vectorized: if `x` is a vector, and so is `v`, then the above returns, in vector format: the probability that a t-distributed variate with `v[1]` degrees of freedom exceeds `x[1]`, the probability that a t-distributed variate with `v[2]` degrees of freedom exceeds `x[2]`, and so on.

```
pt(x, df=v, lower.tail=FALSE)
```

Call `pt()` as in the above line, but replace `x` by the absolute values of the t-statistics you computed for the 9 variables in our data set, and `v` by the degrees of freedom values associated with these t-statistics. Multiply the output by 2, and store it as a vector `pros.dat.p.val`. These are called **p-values** for the t-tests of mean difference between SVI and non-SVI patients, over the 9 variables in our data set. Print out the p-values to the console. Identify the variables for which the p-value is smaller than 0.05 (hence deemed to have a significant difference between SVI and non-SVI patients). Identify the variable with the smallest p-value (the most significant difference between SVI and non-SVI patients).

```
# YOUR CODE GOES HERE
```

Q15. My plot is at your command (optional)

Challenge. Use the last code block as starter code to complete the following task. In the body of a `repeat` loop, prompt the user for a variable name to plot, using `readline()`. Check if the string that you collect from the user is one of the column names in `pros.dat`. Hint: use the `%in%` operator. If the string is indeed one of the column names, produce a histogram of the corresponding variable, with a title and x-axis label set appropriately. If the string is "quit", then break out of the repeat loop. Otherwise, print to the console: "Oops! That's not a variable in my data set." In the Rmd code chunk for your solution code, *make sure to set `eval=FALSE`; otherwise your lab file will never finish knitting.* Try out your solution code by running it in your console.

```
# YOUR CODE GOES HERE
```

Challenge. Extend your prompting code in the last question to allow for scatterplots as well as histograms, and any other options you deem interesting, that the user might want to specify.

```
# YOUR CODE GOES HERE
```

States data set

Below we construct a data frame, of 50 states x 10 variables. The first 8 variables are numeric and the last 2 are factors. The numeric variables here come from the built-in `state.x77` matrix, which records various demographic factors on 50 US states, measured in the 1970s. You can learn more about this state data set by typing `?state.x77` into your R console.

```
state.df = data.frame(state.x77, Region=state.region, Division=state.division)
```

Q16. Basic data frame manipulations

16a. Add a column to `state.df`, containing the state abbreviations that are stored in the built-in vector `state.abb`. Name this column `Abbr`. You can do this in (at least) two ways: by using a call to `data.frame()`, or by directly defining `state.df$Abbr`. Display the first 3 rows and all 11 columns of the new `state.df`.

```
# YOUR CODE GOES HERE
```

16b. Remove the `Region` column from `state.df`. You can do this in (at least) two ways: by using negative indexing, or by directly setting `state.df$Region` to be `NULL`. Display the first 3 rows and all 10 columns of `state.df`.

```
# YOUR CODE GOES HERE
```

17c. Add two columns to `state.df`, containing the x and y coordinates (longitude and latitude, respectively) of the center of the states, that are stored in the (existing) list `state.center`. Hint: take a look at this list in the console, to see what its elements are named. Name these two columns `Center.x` and `Center.y`. Display the first 3 rows and all 12 columns of `state.df`.

```
# YOUR CODE GOES HERE
```

17d. Make a new data frame which contains only those states whose longitude is less than -100. Do this in two different ways: using manual indexing, and `subset()`. Check that they are equal to each other, using an appropriate function call.

```
# YOUR CODE GOES HERE
```

17e. Make a new data frame which contains only the states whose longitude is less than -100, and whose murder rate is above 9%. Print this new data frame to the console. Among the states in this new data frame, which has the highest average life expectancy?

```
# YOUR CODE GOES HERE
```

Prostate cancer data set

Below we read in the prostate cancer data set that we looked in the last lab. You can remind yourself about what's been measured by looking back at the lab.

```
pros.dat =
  read.table("http://www.stat.cmu.edu/~ryantibs/statcomp/data/pros.dat")
```

Q18. Practice with the apply family

18a. Using `sapply()`, calculate the mean of each variable. Also, calculate the standard deviation of each variable. Each should require just one line of code. Display your results.

```
# YOUR CODE GOES HERE
```

18b. Let's plot each variable against SVI. Using `lapply()`, plot each column, excluding SVI, on the y-axis with SVI on the x-axis. This should require just one line of code. **Challenge**: label the y-axes in your plots appropriately. Your solution should still consist of just one line of code and use an apply function. Hint: for this part, consider using `mapply()`.

```
# YOUR CODE GOES HERE
```

18c. Now, use `lapply()` to perform t-tests for each variable in the data set, between SVI and non-SVI groups. To be precise, you will perform a t-test for each variable excluding the SVI variable itself. For convenience, we've defined a function `t.test.by.ind()` below, which takes a numeric variable `x`, and then an indicator variable `ind` (of 0s and 1s) that defines the groups. Run this function on the columns of `pros.dat`, excluding the SVI column itself, and save the result as `tests`. What kind of data structure is `tests`? Print it to the console.

```
t.test.by.ind = function(x, ind) {
  stopifnot(all(ind %in% c(0, 1)))
  return(t.test(x[ind == 0], x[ind == 1]))
}
```

```
# YOUR CODE GOES HERE
```

18d. Using `lapply()` again, extract the p-values from the `tests` object you created in the last question, with just a single line of code. Hint: first, take a look at the first element of `tests`, what kind of object is it, and how is the p-value stored? Second, run the command `` `[[`(pros.dat, "lcavol") `` in your console—what does this do? Now use what you've learned to extract p-values from the `tests` object.

```
# YOUR CODE GOES HERE
```

Rio Olympics data set

Now we're going to examine data from the 2016 Summer Olympics in Rio de Janeiro, taken from https://github.com/flother/rio2016 (complete data on the 2020 Summer Olympics in Tokyo doesn't appear to be available yet). Below we read in the data and store it as `rio`.

```
rio = read.csv("http://www.stat.cmu.edu/~ryantibs/statcomp/data/rio.csv")
```

Q19. More practice with data frames and apply

19a. What kind of object is rio? What are its dimensions and columns names of `rio`? What does each row represent? Is there any missing data?

```
# YOUR CODE GOES HERE
```

19b. Use `rio` to answer the following questions. How many athletes competed in the 2016 Summer Olympics? How many countries were represented? What were these countries, and how many athletes competed for each one? Which country brought the most athletes, and how many was this? Hint: for a factor variable `f`, you can use `table(f)` see how many elements in `f` are in each level of the factor.

```
# YOUR CODE GOES HERE
```

19c. How many medals of each type—gold, silver, bronze—were awarded at this Olympics? Are they equal? Is this result surprising, and can you explain what you are seeing?

```
# YOUR CODE GOES HERE
```

19d. Create a column called `total` which adds the number of gold, silver, and bronze medals for each athlete, and add this column to `rio`. Which athlete had the most number of medals and how many was this? Gold medals? Silver medals? In the case of ties, here, display all the relevant athletes.

```
# YOUR CODE GOES HERE
```

19e. Using `tapply()`, calculate the total medal count for each country. Save the result as `total.by.nat`, and print it to the console. Which country had the most number of medals, and how many was this? How many countries had zero medals?

```
# YOUR CODE GOES HERE
```

19f. Among the countries that had zero medals, which had the most athletes, and how many athletes was this? (Ouch!)

```
# YOUR CODE GOES HERE
```

Q20. Young and old folks

20a. The variable `date_of_birth` contains strings of the date of birth of each athlete. Use the `substr()` function to extract the year of birth for each athlete, and then create a new numeric variable called `age`, equal to 2016 - (the year of birth). (Here we're ignoring days and months for simplicity.) Hint: to extract

the first 4 characters of a string `str`, you can use `substr(str, 1, 4)`. As always, you can also look at the help file for `substr()` for more details.

Add the `age` variable to the `rio` data frame. variable Who is the oldest athlete, and how old is he/she? Youngest athlete, and how old is he/she? In the case of ties, here, display all the relevant athletes.

```
# YOUR CODE GOES HERE
```

20b. Answer the same questions as in the last part, but now only among athletes who won a medal.

```
# YOUR CODE GOES HERE
```

20c. Using a single call to `tapply()`, answer: how old are the youngest and oldest athletes, for each sport?

```
# YOUR CODE GOES HERE
```

20d. You should see that your output from `tapply()` in the last part is a list, which is not particularly convenient. Convert this list into a matrix that has one row for each sport, and two columns that display the ages of the youngest and oldest athletes in that sport. The first 3 rows should look like this:

```
                Youngest Oldest
athletics             14     41
archery               17     44
athletics             16     47
```

You'll notice that we set the row names according to the sports, and we also set appropriate column names. Hint: `unlist()` will unravel all the values in a list; and `matrix()`, as you've seen before, can be used to create a matrix from a vector of values. After you've converted the results to a matrix, print it to the console (and make sure its first 3 rows match those displayed above).

```
# YOUR CODE GOES HERE
```

Challenge. Determine the *names* of the youngest and oldest athletes in each sport, along with their ages (so your result should have 4 columns), without using any explicit iteration. In the case of ties, just return one relevant athlete name. (For this part, you can use another package, such as `plyr` or `dplyr` if you want to.)

```
# YOUR CODE GOES HERE
```

Q21. Sport by sport

21a. Create a new data frame called `sports`, which we'll populate with information about each sporting event at the Summer Olympics. Initially, define `sports` to contain a single variable called `sport` which contains the names of the sporting events in alphabetical order. Then, add a column called `n_participants` which contains the number of participants in each sport. Use one of the apply functions to determine the number of gold medals given out for each sport, and add this as a column called `n_gold`. Using your newly created `sports` data frame, calculate the ratio of the number of gold medals to participants for each sport. Which sport has the highest ratio? Which has the lowest?

```
# YOUR CODE GOES HERE
```

21b. Use one of the apply functions to compute the average weight of the participants in each sport, and add this as a column to `sports` called `ave_weight`. Important: there are missing weights in the data set coded as `NA`, but your column `ave_weight` should ignore these, i.e., it should be itself free of `NA` values. You will have to pass an additional argument to your apply call in order to achieve this. Hint: look at the help file for the `mean()` function; what argument can you set to ignore `NA` values? Once computed, display the average weights along with corresponding sport names, in decreasing order of average weight.

```
# YOUR CODE GOES HERE
```

21c. As in the last part, compute the average weight of athletes in each sport, but now separately for men and women. You should therefore add two new columns, called `ave_weight_men` and `ave_weight_women`, to `sports`. Once computed, display the average weights along with corresponding sports, for men and women, each list sorted in decreasing order of average weight. Are the orderings roughly similar?

```
# YOUR CODE GOES HERE
```

Challenge. Use one of the apply functions to compute the proportion of women among participating athletes in each sport. Use these proportions to recompute the average weight (over all athletes in each sport) from the `ave_weight_men` and `average_weight_women` columns, and define a new column `ave_weight2` accordingly. Does `ave_weight2` differ from `ave_weight`? It should. Explain why. Then show how to recompute the average weight from `ave_weight_men` and `average_weight_women` in a way that exactly recreates `average_weight`.

```
# YOUR CODE GOES HERE
```

Hint #4. Huber loss function

The Huber loss function (or just Huber function, for short) is defined as:

$$\psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases}$$

This function is quadratic on the interval [-1,1], and linear outside of this interval. It transitions from quadratic to linear "smoothly", and looks like this. It is often used in place of the usual squared error loss for robust estimation. For example, the sample average, $\bar{X}$—which given a sample $X_1, \ldots, X_n$ minimizes the squared error loss $\sum_{i=1}^{n}(X_i - m)^2$ over all choices of m—can be inaccurate as an estimate of $\mathbb{E}(X)$ if the distribution of X is heavy-tailed. In such cases, minimizing Huber loss can give a better estimate.

Q22. Some simple function tasks

22a. Write a function `huber()` that takes as an input a number x, and returns the Huber value $\psi(x)$, as defined above. Hint: the body of a function is just a block of R code, e.g., in this code you can use `if()` and `else()` statements. Check that `huber(1)` returns 1, and `huber(4)` returns 7.

```
# YOUR CODE GOES HERE
```

22b. The Huber function can be modified so that the transition from quadratic to linear happens at an arbitrary cutoff value a, as in:

$$\psi_a(x) = \begin{cases} x^2 & \text{if } |x| \leq a \\ 2a|x| - a^2 & \text{if } |x| > a \end{cases}$$

Starting with your solution code to the last question, update your `huber()` function so that it takes two arguments: x, a number at which to evaluate the loss, and a a number representing the cutoff value. It should now return $\psi_a(x)$, as defined above. Check that `huber(3, 2)` returns 8, and `huber(3, 4)` returns 9.

```
# YOUR CODE GOES HERE
```

22c. Update your `huber()` function so that the default value of the cutoff a is 1. Check that `huber(3)` returns 5.

```
# YOUR CODE GOES HERE
```

22d. Check that `huber(a=1, x=3)` returns 5. Check that `huber(1, 3)` returns 1. Explain why these are different.

```
# YOUR CODE GOES HERE
```

22e. Vectorize your `huber()` function, so that the first input can actually be a vector of numbers, and what is returned is a vector whose elements give the Huber evaluated at each of these numbers. Hint: you might try using `ifelse()`, if you haven't already, to vectorize nicely. Check that `huber(x=1:6, a=3)` returns the vector of numbers (1, 4, 9, 15, 21, 27).

```
# YOUR CODE GOES HERE
```

Challenge. Your instructor computed the Huber function values $\psi_a(x)$ over a bunch of different x values, stored in `huber.vals` and `x.vals`, respectively. However, the cutoff a was, let's say, lost. Using `huber.vals`, `x.vals`, and the definition of the Huber function, you should be able to figure out the cutoff value a, at least roughly. Estimate a and explain how you got there. Hint: one way to estimate a is to do so visually, using plotting tools; there are other ways too.

```
# YOUR CODE GOES HERE
```

Q23. Plotting practice, side effects

23a. Professor Tibs created in plot of the Huber function displayed here. Reproduce this plot with your own plotting code, and the `huber()` function you wrote above. The axes and title should be just the same, so should the Huber curve (in black), so should be the red dotted lines at the values -1 and 1, and so should the text "Linear", "Quadratic", "Linear".

```
# YOUR CODE GOES HERE
```

23b. Modify the `huber()` function so that, as a side effect, it prints the string "Invented by the great Swiss statistician Peter Huber!" to the console. Hint: use `cat()`. Call your function on an input of your choosing, to demonstrate this side effect.

```
# YOUR CODE GOES HERE
```

Challenge. Further modify your `huber()` function so that, as another side effect, it produces a plot of Switzerland's national flag. Hint: look up this flag up on Google; it's pretty simple; and you should be able to recreate it with a few calls to `rect()`. Call your function on an input of your choosing, to demonstrate its side effects.

```
# YOUR CODE GOES HERE
```

Q24. Exploring function environments

24a. A modified version of the Huber function is given below. You can see that we've defined the variable `x.squared` in the body of the function to be the square of the input argument `x`. In a separate line of code (outside of the function definition), define the variable `x.squared` to be equal to 999. Then call `huber(x=3)`, and display the value of `x.squared`. What is its value? Is this affected by the function call `huber(x=3)`? It shouldn't be! Reiterate this point with several more lines of code, in which you repeatedly define `x.squared` to be something different (even something non numeric, like a string), and then call `huber(x=3)`, and demonstrate afterwards that the value of `x.squared` hasn't changed.

```
huber = function(x, a=1) {
  x.squared = x^2
  ifelse(abs(x) <= a, x.squared, 2*a*abs(x)-a^2)
}
```

```
# YOUR CODE GOES HERE
```

24b. Similar to the last question, define the variable `a` to be equal to -59.6, then call `huber(x=3, a=2)`, and show that the value of `a` after this function call is unchanged. And repeat a few times with different assignments for the variable `a`, to reiterate this point.

```
# YOUR CODE GOES HERE
```

24c. The previous two questions showed you that a function's body has its own environment in which locally defined variables, like those defined in the body itself, or those defined through inputs to the function, take

priority over those defined outside of the function. However, when a variable referred to the body of a function is *not defined in the local environment*, the default is to look for it in the global environment (outside of the function).

Below is a "sloppy" implementation of the Huber function called `huber.sloppy()`, in which the cutoff `a` is not passed as an argument to the function. In a separate line of code (outside of the function definition), define `a` to be equal to 1.5 and then call `huber.sloppy(x=3)`. What is the output? Explain. Repeat this a few times, by defining `a` and then calling `huber.sloppy(x=3)`, to show that the value of `a` does indeed affect the function's ouptut as expected.

Challenge: try setting `a` equal to a string and calling `huber.sloppy(x=3)`. Can you explain what is happening?

```
huber.sloppy = function(x) {
  ifelse(abs(x) <= a, x^2, 2*a*abs(x)-a^2)
}
```

```
# YOUR CODE GOES HERE
```

24d. At last, a difference between `=` and `<-`, explained! Some of you have been asking about this. The equal sign `=` and assignment operator `<-` are often used interchangeably in R, and some people will often say that a choice between the two is mostly a matter of stylistic taste. This is not the full story. Indeed, `=` and `<-` behave very differently when used to set input arguments in a function call. As we showed above, setting, say, `a=5` as the input to `huber()` has no effect on the global assignment for `a`. However, replacing `a=5` with `a<-5` in the call to `huber()` is entirely different in terms of its effect on `a`. Demonstrate this, and explain what you are seeing in terms of global assignment.

```
# YOUR CODE GOES HERE
```

24e. The story now gets even more subtle. It turns out that the assignment operator `<-` allows us to define new global variables even when we are specifying inputs to a function. Pick a variable name that has not been defined yet in your workspace, say `b` (or something else, if this has already been used in your R Markdown document). Call `huber(x=3, b<-20)`, then display the value of `b`—this variable should now exist in the global enviroment, and it should be equal to 20! Also, can you explain the output of `huber(x=3, b<-20)`?

```
# YOUR CODE GOES HERE
```

Challenge. The property of the assignment operator `<-` demonstrated in the last question, although tricky, can also be pretty useful. Leverage this property to plot the function $y = 0.05x^2 - \sin(x)\cos(x) + 0.1\exp(1 + \log(x))$ over 50 x values between 0 and 2, using only one line of code and one call to the function `seq()`.

```
# YOUR CODE GOES HERE
```

Challenge. Give an example to show that the property of the assignment operator `<-` demonstrated in the last two questions does not hold in the body of a function. That is, give an example in which `<-` is used in the body of a function to define a variable, but this doesn't translate into global assignment.

```
# YOUR CODE GOES HERE
```