Lab 2

Janine Lim

11:59PM February 25, 2021

More Basic R Skills

- Create a function `my_reverse` which takes as required input a vector and returns the vector in reverse where the first entry is the last entry, etc. No function calls are allowed inside your function otherwise that would defeat the purpose of the exercise! (Yes, there is a base R function that does this called `rev`). Use `head` on `v` and `tail` on `my_reverse(v)` to verify it works.

```
v = 1:10
my_reverse = function(v){
  v_rev = rep(NA, times = length(v))
  for (i in length(v):1) {
    v_rev[length(v)-i+1] = v[i]
  }
  v_rev
}
my_reverse(v)
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

- Create a function `flip_matrix` which takes as required input a matrix, an argument `dim_to_rev` that returns the matrix with the rows in reverse order or the columns in reverse order depending on the `dim_to_rev` argument. Let the default be the dimension of the matrix that is greater.

```
flip_matrix=function(X, dim_to_rev=NULL){
  if(is.null(dim_to_rev)){
    dim_to_rev = ifelse(nrow(X)>ncol(X), "rows", "cols" )
  }
  if(dim_to_rev =="rows"){
    X[my_reverse(1:nrow(X)), ]
  } else if(dim_to_rev=="cols"){
    X[, my_reverse(1:ncol(X))]
  } else {
    stop("Illegal arg")}
}

X = matrix(rnorm(100), nrow = 25)
X
```

```
##              [,1]        [,2]        [,3]        [,4]
```

```
##  [1,] -0.01963863  0.51234953 -0.57524016 -0.15179440
##  [2,] -0.96147726  0.27477456  0.68054598 -0.12615244
##  [3,] -0.18719207  1.38672449 -0.10647373  0.32412639
##  [4,]  2.39999342  2.32522231  0.36698425 -0.31212609
##  [5,] -0.39409967 -1.62754407 -0.45234157 -1.19845414
##  [6,]  0.88179531  0.69123283  1.04539324 -0.78333969
##  [7,] -0.21802748  1.36683399 -0.69759628  2.29537431
##  [8,] -1.81495589  0.14088880  1.39285791 -0.28656417
##  [9,]  1.65014374 -0.12880740  1.25804484 -1.36949527
## [10,]  1.31505332 -0.04126384  0.20515880 -1.63386246
## [11,] -0.40632811 -0.92094492  0.03788771  0.02699189
## [12,] -1.31333491 -0.05376135  0.14442010 -0.57347494
## [13,]  1.20677625  1.10764942 -0.88139748  0.04217807
## [14,] -0.25892081  0.36656207  0.41797489  0.82543015
## [15,] -0.89896724 -0.39412249  0.38740437 -1.49196464
## [16,]  0.75840388 -0.63980469 -0.78508284  0.11400692
## [17,] -0.85087088 -1.13017161 -1.23191230 -0.53103583
## [18,]  0.51620417  0.44035424 -0.81229860 -0.39863403
## [19,]  0.22558313 -0.31312441 -1.50091470 -0.38460564
## [20,] -0.72023784  0.73575038  2.15651287  0.31453822
## [21,] -1.14173762 -0.31824495  0.19452935  2.02324194
## [22,] -0.04335214  1.68939385  0.04500769  0.55991092
## [23,]  0.30839641  0.05453670 -1.44857104 -1.52022174
## [24,]  1.94158392 -1.03911557 -0.99881512  0.61398503
## [25,] -0.13682452 -1.00445099 -1.06424305  0.35580454
```

```
flip_matrix(X, dim_to_rev = "cols")
```

```
##              [,1]        [,2]        [,3]        [,4]
##  [1,] -0.15179440 -0.57524016  0.51234953 -0.01963863
##  [2,] -0.12615244  0.68054598  0.27477456 -0.96147726
##  [3,]  0.32412639 -0.10647373  1.38672449 -0.18719207
##  [4,] -0.31212609  0.36698425  2.32522231  2.39999342
##  [5,] -1.19845414 -0.45234157 -1.62754407 -0.39409967
##  [6,] -0.78333969  1.04539324  0.69123283  0.88179531
##  [7,]  2.29537431 -0.69759628  1.36683399 -0.21802748
##  [8,] -0.28656417  1.39285791  0.14088880 -1.81495589
##  [9,] -1.36949527  1.25804484 -0.12880740  1.65014374
## [10,] -1.63386246  0.20515880 -0.04126384  1.31505332
## [11,]  0.02699189  0.03788771 -0.92094492 -0.40632811
## [12,] -0.57347494  0.14442010 -0.05376135 -1.31333491
## [13,]  0.04217807 -0.88139748  1.10764942  1.20677625
## [14,]  0.82543015  0.41797489  0.36656207 -0.25892081
## [15,] -1.49196464  0.38740437 -0.39412249 -0.89896724
## [16,]  0.11400692 -0.78508284 -0.63980469  0.75840388
## [17,] -0.53103583 -1.23191230 -1.13017161 -0.85087088
## [18,] -0.39863403 -0.81229860  0.44035424  0.51620417
## [19,] -0.38460564 -1.50091470 -0.31312441  0.22558313
## [20,]  0.31453822  2.15651287  0.73575038 -0.72023784
## [21,]  2.02324194  0.19452935 -0.31824495 -1.14173762
## [22,]  0.55991092  0.04500769  1.68939385 -0.04335214
## [23,] -1.52022174 -1.44857104  0.05453670  0.30839641
## [24,]  0.61398503 -0.99881512 -1.03911557  1.94158392
## [25,]  0.35580454 -1.06424305 -1.00445099 -0.13682452
```

- Create a list named `my_list` with keys "A", "B", … where the entries are arrays of size 1, 2 x 2, 3 x 3 x 3, etc. Fill the array with the numbers 1, 2, 3, etc. Make 8 entries according to this sequence.

```
my_list =list()
for (i in 1:8) {
  my_list[[letters[i]]] = array(1:i^i, dim = c(rep(i , i)))
}
```

Run the following code:

```
lapply(my_list, object.size)
```

```
## $a
## 224 bytes
##
## $b
## 232 bytes
##
## $c
## 352 bytes
##
## $d
## 1248 bytes
##
## $e
## 12744 bytes
##
## $f
## 186864 bytes
##
## $g
## 3294416 bytes
##
## $h
## 67109104 bytes
```

Use `?object.size` to read about what these functions do. Then explain the output you see above. For the later arrays, does it make sense given the dimensions of the arrays?

When we use object.size on my_list, it provides an estimate of the memory used for each of the 8 arrays in our list. It makes sense that the later arrays use more memory because they have larger dimensions and use more memory to store the information.

Now cleanup the namespace by deleting all stored objects and functions:

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

A little about strings

- Use the `strsplit` function and `sample` to put the sentences in the string `lorem` below in random order. You will also need to manipulate the output of `strsplit` which is a list. You may need to learn basic concepts of regular expressions.

```
lorem = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi posuere varius volutpat. Morbi
sample(unlist(strsplit(lorem, split = "[.] ")))
```

```
##  [1] "Donec vehicula sagittis nisi non semper"
##  [2] "Cras suscipit id nibh lacinia elementum"
##  [3] "Integer dapibus mi lectus, eu posuere arcu ultricies in"
##  [4] "Morbi posuere varius volutpat"
##  [5] "Donec at tempor erat"
##  [6] "Curabitur est augue, congue eget quam in, scelerisque semper magna"
##  [7] "Mauris at sodales augue"
##  [8] "Lorem ipsum dolor sit amet, consectetur adipiscing elit"
##  [9] "Morbi faucibus ligula id massa ultricies viverra"
## [10] "Aenean nulla ante, iaculis sed vehicula ac, finibus vel arcu"
```

You have a set of names divided by gender (M / F) and generation (Boomer / GenX / Millenial):

- M / Boomer "Theodore, Bernard, Gene, Herbert, Ray, Tom, Lee, Alfred, Leroy, Eddie"
- M / GenX "Marc, Jamie, Greg, Darryl, Tim, Dean, Jon, Chris, Troy, Jeff"
- M / Millennial "Zachary, Dylan, Christian, Wesley, Seth, Austin, Gabriel, Evan, Casey, Luis"
- F / Boomer "Gloria, Joan, Dorothy, Shirley, Betty, Dianne, Kay, Marjorie, Lorraine, Mildred"
- F / GenX "Tracy, Dawn, Tina, Tammy, Melinda, Tamara, Tracey, Colleen, Sherri, Heidi"
- F / Millennial "Samantha, Alexis, Brittany, Lauren, Taylor, Bethany, Latoya, Candice, Brittney, Cheyenne"

Create a list-within-a-list that will intelligently store this data.

```
#HINT:
gen_list = list("male" = list(), "female" = list())
gen_list$male$Boomer = strsplit("Theodore, Bernard, Gene, Herbert, Ray, Tom, Lee, Alfred, Leroy, Eddie"
gen_list$male$GenX = strsplit("Marc, Jamie, Greg, Darryl, Tim, Dean, Jon, Chris, Troy, Jeff", split = "
gen_list$male$Millennial = strsplit("Zachary, Dylan, Christian, Wesley, Seth, Austin, Gabriel, Evan, Ca
gen_list$female$Boomer = strsplit("Gloria, Joan, Dorothy, Shirley, Betty, Dianne, Kay, Marjorie, Lorrai
gen_list$female$GenX= strsplit("Tracy, Dawn, Tina, Tammy, Melinda, Tamara, Tracey, Colleen, Sherri, Hei
gen_list$female$Millennial = strsplit("Samantha, Alexis, Brittany, Lauren, Taylor, Bethany, Latoya, Can
```

Dataframe creation

Imagine you are running an experiment with many manipulations. You have 14 levels in the variable "treatment" with levels a, b, c, etc. For each of those manipulations you have 3 submanipulations in a variable named "variation" with levels A, B, C. Then you have "gender" with levels M / F. Then you have "generation" with levels Boomer, GenX, Millenial. Then you will have 6 runs per each of these groups. In each set of 6 you will need to select a name without duplication from the appropriate set of names (from the last question). Create a data frame with columns treatment, variation, gender, generation, name and y that will store all the unique unit information in this experiment. Leave y empty because it will be measured as the experiment is executed.

```
n = 14 * 3 * 2 * 3 * 10
X = data.frame(treatment = sort(rep(letters[1:14], each = n/14)),
               variation = (rep(LETTERS[1:3], each = n/14/3)),
               gender = rep(c("M", "F"),each =n/14/3/2),
               generation = rep(c("Boomer", "GenX", "Millennial"), each = n/14/3/2/3),
```

```
                name = rep(unlist(gen_list), times = 14*3),
                y = rep(" ", n)
                )
X
```

Packages

Install the package `pacman` using regular base R.

```
install.packages("pacman")
```

First, install the package `testthat` (a widely accepted testing suite for R) from https://github.com/r-lib/testthat using `pacman`. If you are using Windows, this will be a long install, but you have to go through it for some of the stuff we are doing in class. LINUX (or MAC) is preferred for coding. If you can't get it to work, install this package from CRAN (still using `pacman`), but this is not recommended long term.

```
pacman::p_load(testthat)
```

- Create vector `v` consisting of all numbers from -100 to 100 and test using the second line of code su

```
v= seq(-100, 100)
expect_equal(v, -100 : 101)
```

If there are any errors, the `expect_equal` function will tell you about them. If there are no errors, then it will be silent.

Test the `my_reverse` function from lab2 using the following code:

```
v=1:100
expect_equal(my_reverse(v), rev(v))
expect_equal(my_reverse(c("A", "B", "C")), c("C", "B", "A"))
```

Multinomial Classification using KNN

Write a $k = 1$ nearest neighbor algorithm using the Euclidean distance function. This is standard "Roxygen" format for documentation. Hopefully, we will get to packages at some point and we will go over this again. It is your job also to fill in this documentation.

```
#' 1 nearest neighbor classifier
#'
#' Classify an observation based on the label of the closest observation in a set of training observati
#'
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    The vector of training data labels
#' @param xtest       A test observation as a row vector
#' @return            This returns the predicted binary value for the test observation
nn_algorithm_predict = function(Xinput, y_binary, xtest){
  n=nrow(Xinput)
  distances = array(NA,n)
  for (i in 1:n) {
```

```
    distances[i]=sum((Xinput[i,]-xtest)^2)
  }
  which.min(distances)
  y_binary[which.min(distances)]
}
```

Write a few tests to ensure it actually works:

```
set.seed(11)
fake_data = matrix(NA, nrow = 100, ncol = 3)
fake_data[,1] = sample(rnorm(n=100, mean=0, sd=1), replace=TRUE)
fake_data[,2]=sample(rnorm(n=100, mean =0, sd=1), replace=TRUE)
fake_data[,3]=sample(rep(c(0,1), each = 50))
head(fake_data)
```

```
##             [,1]        [,2] [,3]
## [1,]  0.31549629 -1.63889607    1
## [2,]  0.48467705 -0.26582319    0
## [3,]  0.02038144 -0.82962083    1
## [4,]  1.17848916  0.93892206    0
## [5,] -0.68877213  0.36638081    0
## [6,]  0.75440166  0.04290007    0
```

```
x_star1 = c(-0.12, 2.85)
x_star2 = c(1.30, -0.61)
x_star3 = c(0.78, 1.34)

nn_algorithm_predict(fake_data[,1:2], fake_data[,3], x_star1)
```

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

```
nn_algorithm_predict(fake_data[,1:2], fake_data[,3], x_star2)
```

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

```
nn_algorithm_predict(fake_data[,1:2], fake_data[,3], x_star3)
```

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

We now add an argument `d` representing any legal distance function to the `nn_algorithm_predict` function. Update the implementation so it performs NN using that distance function. Set the default function to be the Euclidean distance in the original function. Also, alter the documentation in the appropriate places.

```
#' 1 nearest neighbor classifier
#'
#' Classify an observation based on the label of the closest observation in a set of training observati
#'
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    The vector of training data labels
#' @param xtest       A test observation as a row vector
```

```
#' @param d           A distance function which takes 2 row vectors as inputs
#' @return            This returns the predicted binary value for the test observation
nn_algorithm_predict = function(Xinput, y_binary, xtest, d=function(v1, v2){sum((v1-v2)^2)}){
  n=nrow(Xinput)
  distances = array(NA,n)
  for (i in 1:n) {
    distances[i]=d(Xinput[i,], xtest)
  }
  which.min(distances)
  y_binary[which.min(distances)]
}
```

For extra credit (unless you're a masters student), add an argument `k` to the `nn_algorithm_predict` function and update the implementation so it performs KNN. In the case of a tie, choose $\hat{y}$ randomly. Set the default `k` to be the square root of the size of $\mathcal{D}$ which is an empirical rule-of-thumb popularized by the "Pattern Classification" book by Duda, Hart and Stork (2007). Also, alter the documentation in the appropriate places.

```
#' K nearest neighbor classifier
#'
#' Classify an observation based on the label of the closest observation in a set of training observati
#'
#' @param k           The number of closest observations we want to use to predict on our test observat
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    The vector of training data labels for a binary output variable.
#' @param xtest       A test observation as a row vector
#' @param d           A distance function which takes 2 row vectors as inputs
#' @return            This returns the predicted binary value for the test observation
nn_algorithm_predict = function(Xinput, y_binary, xtest, d=function(v1, v2){sum((v1-v2)^2)}, k = NULL )
  n=nrow(Xinput)
  distances = array(NA,n)
 if(is.null(k)){ #make k the sqrt of D if null
   k = sqrt(n)
 }
  for (i in 1:n) {
    distances[i]=d(Xinput[i,], xtest)
  }
  unique_dist = sort(unique(distances), decreasing = FALSE)
  k_dist = unique_dist[1:k] #grabs the k closest distances
for (j in 1:k){ #find the label for each k observation
  k_labels = array(NA, k)
  k_labels[j] = y_binary[k_dist[j]]
}
  tab_k_labs = table(k_labels)
  if (tab_k_labs[1]!=tab_k_labs[2]) { #if the count for both labels are not the same
    lab = names(sort(tab_k_labs, decreasing = TRUE)[1]) #gets the label for the mode of the table of la
   y_star = y_binary[lab]
  }
  else {
    y_star = y_binary[sample(k_labels, size = 1)]
  }
  y_star
}
```

Basic Binary Classification Modeling

- Load the famous `iris` data frame into the namespace. Provide a summary of the columns using the `skim` function in package `skimr` and write a few descriptive sentences about the distributions using the code below and in English.

```
data(iris)
pacman::p_load(skimr)
skim(iris)
```

Table 1: Data summary

Name	iris
Number of rows	150
Number of columns	5
Column type frequency:	
factor	1
numeric	4
Group variables	None

Variable type: factor

skim_variable	n_missing	complete_rate	ordered	n_unique	top_counts
Species	0	1	FALSE	3	set: 50, ver: 50, vir: 50

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
Sepal.Length	0	1	5.84	0.83	4.3	5.1	5.80	6.4	7.9	
Sepal.Width	0	1	3.06	0.44	2.0	2.8	3.00	3.3	4.4	
Petal.Length	0	1	3.76	1.77	1.0	1.6	4.35	5.1	6.9	
Petal.Width	0	1	1.20	0.76	0.1	0.3	1.30	1.8	2.5	

The iris dataset has 1 factor variable with 3 levels and 4 numeric variables, acting as the features/inputs and 150 observations, with no missing observations. Each numeric variable has low SD and so observations are all relatively close to the mean.

The outcome / label / response is `Species`. This is what we will be trying to predict. However, we only care about binary classification between "setosa" and "versicolor" for the purposes of this exercise. Thus the first order of business is to drop one class. Let's drop the data for the level "virginica" from the data frame.

```
iris = iris[iris$Species != "virginica", ]
```

Now create a vector `y` that is length the number of remaining rows in the data frame whose entries are 0 if "setosa" and 1 if "versicolor".

```
y = as.numeric(iris$Species =="versicolor")
head(cbind(iris, y)) ##double check to make sure setosa = 0 and versicolor = 1
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species y
## 1          5.1         3.5          1.4         0.2  setosa 0
## 2          4.9         3.0          1.4         0.2  setosa 0
## 3          4.7         3.2          1.3         0.2  setosa 0
## 4          4.6         3.1          1.5         0.2  setosa 0
## 5          5.0         3.6          1.4         0.2  setosa 0
## 6          5.4         3.9          1.7         0.4  setosa 0
```

```
tail(cbind(iris, y))
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species y
## 95           5.6         2.7          4.2         1.3 versicolor 1
## 96           5.7         3.0          4.2         1.2 versicolor 1
## 97           5.7         2.9          4.2         1.3 versicolor 1
## 98           6.2         2.9          4.3         1.3 versicolor 1
## 99           5.1         2.5          3.0         1.1 versicolor 1
## 100          5.7         2.8          4.1         1.3 versicolor 1
```

- Write a function `mode` returning the sample mode.

```
mode = function(v){
  names(sort(table(v), decreasing = TRUE)[1])
}
mode(sample(letters, 1000, replace=TRUE))
```

```
## [1] "p"
```

- Fit a threshold model to `y` using the feature `Sepal.Length`. Write your own code to do this. What is the estimated value of the threshold parameter? Save the threshold value as `threshold`. The estimated value of the threshold parameter is 5.4, with 11 errors.

```
n = nrow(iris)
num_errors_by_parameter = matrix(NA, nrow = n, ncol = 2)
colnames(num_errors_by_parameter) = c("threshold_parameter", "num_errors")
y
```

```
##   [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
##  [36] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
for (i in 1:n){
  threshold = iris$Sepal.Length[i]
  num_errors = sum((iris$Sepal.Length>threshold) !=y)
  num_errors_by_parameter[i, ] =c(threshold, num_errors)
}

num_errors_by_parameter[order(num_errors_by_parameter[, "num_errors"]), ]
```

```
##        threshold_parameter num_errors
##   [1,]                 5.4         11
##   [2,]                 5.4         11
##   [3,]                 5.4         11
##   [4,]                 5.4         11
##   [5,]                 5.4         11
##   [6,]                 5.4         11
##   [7,]                 5.5         14
##   [8,]                 5.5         14
##   [9,]                 5.5         14
##  [10,]                 5.5         14
##  [11,]                 5.5         14
##  [12,]                 5.5         14
##  [13,]                 5.5         14
##  [14,]                 5.3         15
##  [15,]                 5.2         16
##  [16,]                 5.2         16
##  [17,]                 5.2         16
##  [18,]                 5.2         16
##  [19,]                 5.1         18
##  [20,]                 5.1         18
##  [21,]                 5.1         18
##  [22,]                 5.1         18
##  [23,]                 5.1         18
##  [24,]                 5.1         18
##  [25,]                 5.1         18
##  [26,]                 5.1         18
##  [27,]                 5.1         18
##  [28,]                 5.6         19
##  [29,]                 5.6         19
##  [30,]                 5.6         19
##  [31,]                 5.6         19
##  [32,]                 5.6         19
##  [33,]                 5.7         22
##  [34,]                 5.7         22
##  [35,]                 5.7         22
##  [36,]                 5.7         22
##  [37,]                 5.7         22
##  [38,]                 5.7         22
##  [39,]                 5.7         22
##  [40,]                 5.8         24
##  [41,]                 5.8         24
##  [42,]                 5.8         24
##  [43,]                 5.8         24
##  [44,]                 5.0         25
##  [45,]                 5.0         25
##  [46,]                 5.0         25
##  [47,]                 5.0         25
##  [48,]                 5.0         25
##  [49,]                 5.0         25
##  [50,]                 5.0         25
##  [51,]                 5.0         25
##  [52,]                 5.0         25
##  [53,]                 5.0         25
```

```
##  [54,]                5.9         26
##  [55,]                5.9         26
##  [56,]                6.0         30
##  [57,]                6.0         30
##  [58,]                6.0         30
##  [59,]                6.0         30
##  [60,]                4.9         31
##  [61,]                4.9         31
##  [62,]                4.9         31
##  [63,]                4.9         31
##  [64,]                4.9         31
##  [65,]                4.8         34
##  [66,]                4.8         34
##  [67,]                4.8         34
##  [68,]                4.8         34
##  [69,]                4.8         34
##  [70,]                6.1         34
##  [71,]                6.1         34
##  [72,]                6.1         34
##  [73,]                6.1         34
##  [74,]                6.2         36
##  [75,]                6.2         36
##  [76,]                4.7         39
##  [77,]                4.7         39
##  [78,]                6.3         39
##  [79,]                6.3         39
##  [80,]                6.3         39
##  [81,]                4.6         41
##  [82,]                4.6         41
##  [83,]                4.6         41
##  [84,]                4.6         41
##  [85,]                6.4         41
##  [86,]                6.4         41
##  [87,]                6.5         42
##  [88,]                6.6         44
##  [89,]                6.6         44
##  [90,]                4.5         45
##  [91,]                4.4         46
##  [92,]                4.4         46
##  [93,]                4.4         46
##  [94,]                6.7         47
##  [95,]                6.7         47
##  [96,]                6.7         47
##  [97,]                6.8         48
##  [98,]                4.3         49
##  [99,]                6.9         49
## [100,]                7.0         50
```

```
best_row = order(num_errors_by_parameter[, "num_errors"])[1]
threshold = c(num_errors_by_parameter[best_row, "threshold_parameter"], use.names = FALSE)
threshold
```

```
## [1] 5.4
```

What is the total number of errors this model makes?

The total number of errors is 2796.

```
sum(num_errors_by_parameter[,2])
```

```
## [1] 2796
```

Does the threshold model's performance make sense given the following summaries:

```
threshold
```

```
## [1] 5.4
```

```
summary(iris[iris$Species == "setosa", "Sepal.Length"])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   4.800   5.000   5.006   5.200   5.800
```

```
summary(iris[iris$Species == "versicolor", "Sepal.Length"])
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.900   5.600   5.900   5.936   6.300   7.000
```

Yes this makes sense because the threshold parameter is approximately halfway between the mean of the sepal length for the setosa flowers and the mean of the sepal length of the versicolor flowers, meaning there will still be some error but should capture at approximately 3/4 of the correct predictions for both setosa and versicolor.

Create the function `g` explicitly that can predict `y` from `x` being a new `Sepal.Length`.

```
g = function(x){
  ifelse(x > threshold, 1, 0)
}
```

Perceptron

You will code the "perceptron learning algorithm" for arbitrary number of features p. Take a look at the comments above the function. Respect the spec below:

```
#' The Perceptron Algorithm for p-features
#'
#' The Perceptron takes linearly separable data and finds the best wedge and median line to separate th
#'
#' @param Xinput      A matrix of training data with p columns that represent p continuous features
#' @param y_binary    This will be a vector with the binary labels from the training/input data
#' @param MAX_ITER    A pre-specified number of iterations to run through to try and minimize misclassi
#' @param w           An initialized p+1 dimensional vector
#'
#' @return            The computed final parameter (weight) as a vector of length p + 1
```

```
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w = NULL){

Xinput1 = as.matrix(cbind(1,Xinput))
p = ncol(Xinput1)
w = rep(0, p)

for (iter in 1 : MAX_ITER){
  for (i in 1 : nrow(Xinput1)) {
    x_i = Xinput1[i, ]
    yhat_i = ifelse(sum(x_i * w) > 0, 1, 0)
    y_i = y_binary[i]
    for(j in 1:p){
      w[j] = w[j] + (y_i - yhat_i) * x_i[j]
    }
  }
}
w
}
```

To understand what the algorithm is doing - linear "discrimination" between two response categories, we can draw a picture. First let's make up some very simple training data $\mathbb{D}$.

```
Xy_simple = data.frame(
 response = factor(c(0, 0, 0, 1, 1, 1)), #nominal
 first_feature = c(1, 1, 2, 3, 3, 4),    #continuous
 second_feature = c(1, 2, 1, 3, 4, 3)    #continuous
)
```

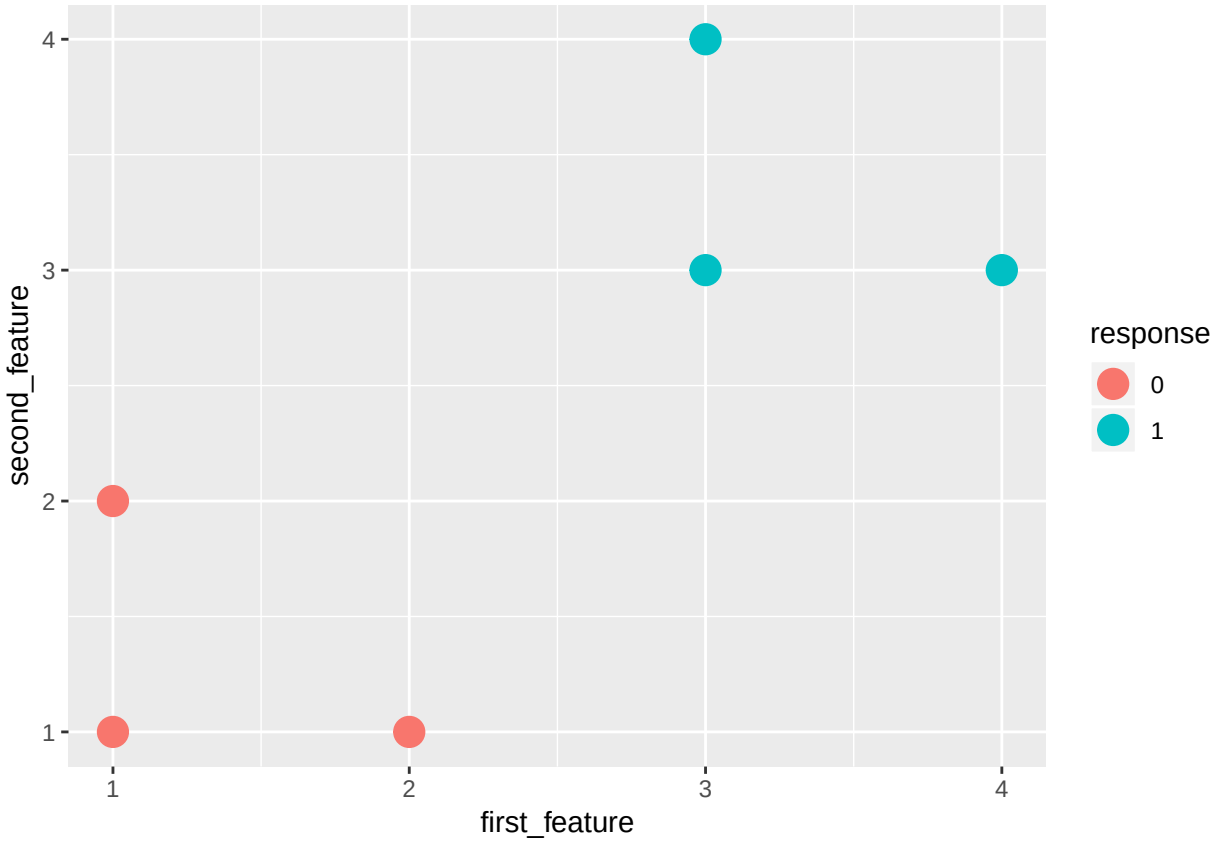
We haven't spoken about visualization yet, but it is important we do some of it now. Thus, I will write this code for you and you will just run it. First we load the visualization library we're going to use:

```
pacman::p_load(ggplot2)
```

We are going to just get some plots and not talk about the code to generate them as we will have a whole unit on visualization using `ggplot2` in the future.

Let's first plot y by the two features so the coordinate plane will be the two features and we use different colors to represent the third dimension, y.

```
simple_viz_obj = ggplot(Xy_simple, aes(x = first_feature, y = second_feature, color = response)) +
  geom_point(size = 5)
simple_viz_obj
```



We are plotting first feature by the second feature and differentiating between the response by color.

Now, let us run the algorithm and see what happens:

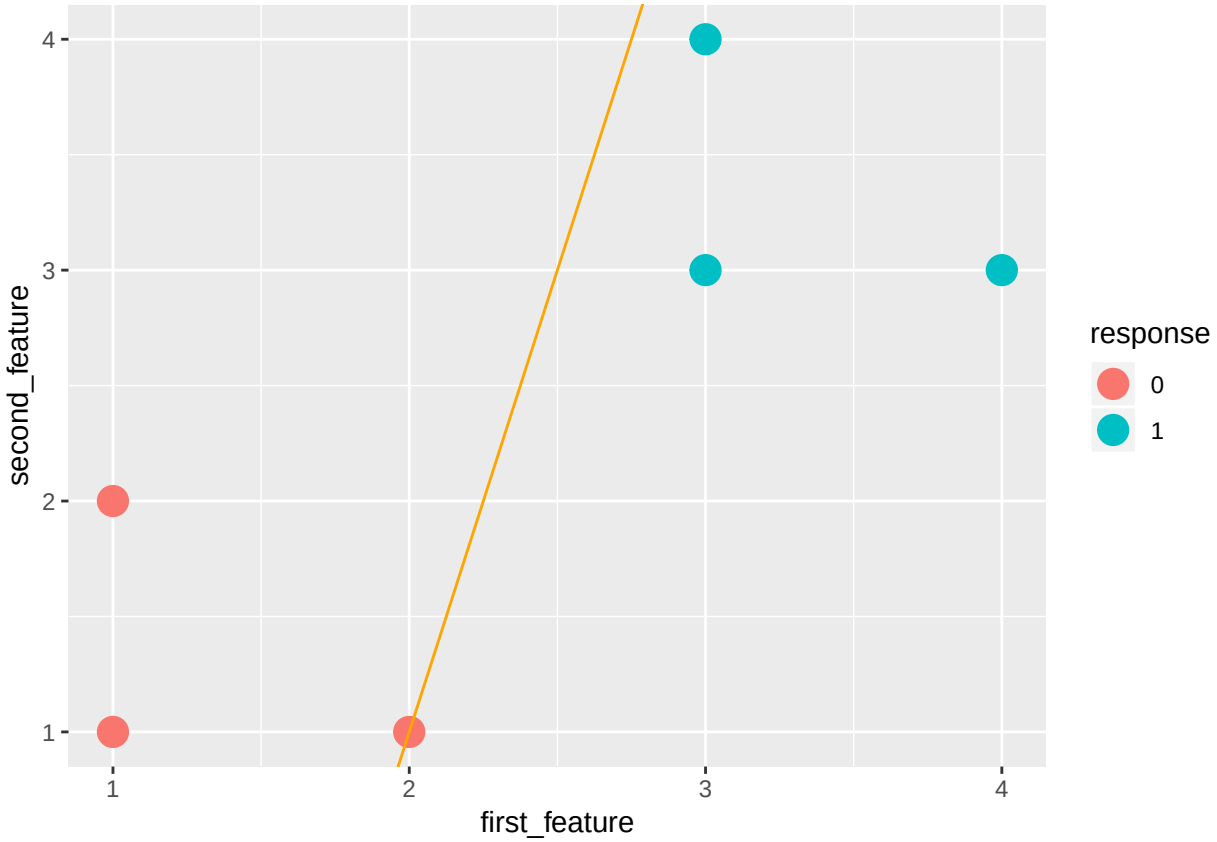
```
w_vec_simple_per = perceptron_learning_algorithm(
  cbind(Xy_simple$first_feature, Xy_simple$second_feature),
  as.numeric(Xy_simple$response == 1))
w_vec_simple_per
```

```
## [1] -7  4 -1
```

Explain this output. What do the numbers mean? What is the intercept of this line and the slope? You will have to do some algebra.

The numbers tell us how to draw a line that separates the data based on the response variable. The first number is the y-intercept, the second number is the coefficient for the first feature, and the third number is the coefficient for the second feature. So we see that the intercept is -7 and the slope is 4.

```
simple_perceptron_line = geom_abline(
    intercept = -w_vec_simple_per[1] / w_vec_simple_per[3],
    slope = -w_vec_simple_per[2] / w_vec_simple_per[3],
    color = "orange")
simple_viz_obj + simple_perceptron_line
```



Explain this picture. Why is this line of separation not "satisfying" to you?

This line of separation is highly unsatisfying because we can clearly see the data is linearly separable, and it does a very poor job at separating the data points.

For extra credit, program the maximum-margin hyperplane perceptron that provides the best linear discrimination model for linearly separable data. Make sure you provide ROxygen documentation for this function.

```
#TO-DO
```