Lab 2

Frank Palma Gomez

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.

```
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] # reverse index
  }

  v_rev
}

v =  1:10

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 argument")
  }
}

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

```
##                [,1]        [,2]        [,3]        [,4]
##  [1,] -0.14595082  0.12150545 -1.27315564  0.87223502
##  [2,]  0.56599714  1.51006194  2.07274756  1.83043355
##  [3,]  0.66941785  0.90587485  1.41811272  0.22505885
##  [4,]  0.68474011  0.52666583  1.11672904 -0.79902499
##  [5,] -0.44667630 -1.32493340  0.86004003 -0.92289270
##  [6,]  0.59337377 -2.37543149  0.67971869 -0.36265600
##  [7,]  0.09353828 -1.67155954  0.28205905  0.39181293
##  [8,]  2.11397779  1.04220388  0.88824640 -0.50357679
##  [9,]  0.61664783  0.23794877 -0.30488653  3.00208773
## [10,]  0.10084593 -0.97005954 -0.22847586  0.46096534
## [11,]  0.83674216  0.69193296 -1.70930890  1.52931527
## [12,]  1.22248917  0.50256136  0.68889536  0.94257443
## [13,] -0.79311993 -0.44795843 -1.39944314 -0.08952538
## [14,] -2.01446205 -3.36223524 -0.03247583  1.21634669
## [15,] -0.67877293 -0.98082998  0.55938646  0.69399704
## [16,] -0.11597497  0.20107621 -0.06748160  0.12558354
## [17,] -1.79544252  1.42343524  0.41972161  1.31642031
## [18,] -0.58459697  0.23200275 -0.55536229  0.67303138
## [19,]  1.29783707  1.40346055  1.47690036  0.93787489
## [20,]  0.99345735 -0.01236711  0.68374436 -0.75588129
## [21,]  0.32926201 -0.39650542  0.01976943  0.30864351
## [22,]  0.86677799 -0.29531376 -2.20624292  0.69677904
## [23,] -1.22323485  1.28562793  0.22875781 -0.86639869
## [24,] -0.19224649  0.68460043  0.00620631  0.73656577
## [25,] -0.75019073  0.66949650  0.41458835  1.26068895
```

```
flip_matrix(X)
```

```
##              [,1]        [,2]        [,3]        [,4]
##  [1,] -0.75019073  0.66949650  0.41458835  1.26068895
##  [2,] -0.19224649  0.68460043  0.00620631  0.73656577
##  [3,] -1.22323485  1.28562793  0.22875781 -0.86639869
##  [4,]  0.86677799 -0.29531376 -2.20624292  0.69677904
##  [5,]  0.32926201 -0.39650542  0.01976943  0.30864351
##  [6,]  0.99345735 -0.01236711  0.68374436 -0.75588129
##  [7,]  1.29783707  1.40346055  1.47690036  0.93787489
##  [8,] -0.58459697  0.23200275 -0.55536229  0.67303138
##  [9,] -1.79544252  1.42343524  0.41972161  1.31642031
## [10,] -0.11597497  0.20107621 -0.06748160  0.12558354
## [11,] -0.67877293 -0.98082998  0.55938646  0.69399704
## [12,] -2.01446205 -3.36223524 -0.03247583  1.21634669
## [13,] -0.79311993 -0.44795843 -1.39944314 -0.08952538
## [14,]  1.22248917  0.50256136  0.68889536  0.94257443
## [15,]  0.83674216  0.69193296 -1.70930890  1.52931527
## [16,]  0.10084593 -0.97005954 -0.22847586  0.46096534
## [17,]  0.61664783  0.23794877 -0.30488653  3.00208773
## [18,]  2.11397779  1.04220388  0.88824640 -0.50357679
## [19,]  0.09353828 -1.67155954  0.28205905  0.39181293
## [20,]  0.59337377 -2.37543149  0.67971869 -0.36265600
## [21,] -0.44667630 -1.32493340  0.86004003 -0.92289270
## [22,]  0.68474011  0.52666583  1.11672904 -0.79902499
## [23,]  0.66941785  0.90587485  1.41811272  0.22505885
## [24,]  0.56599714  1.51006194  2.07274756  1.83043355
## [25,] -0.14595082  0.12150545 -1.27315564  0.87223502
```

- 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()
letter_list = LETTERS[1:8]
for (letter in letter_list) {
  letter_idx = which(letter == LETTERS)
  len = letter_idx * letter_idx
  my_list[[letter]] = array(1:len, dim=rep(letter_idx, letter_idx))
}

head(my_list$A)
```

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

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?

object.size calculates what the size of an object is. You can see the byte size increase exponentially since we are storing high dimensional arrays. The later arrays make sense since we have an array with dimensions 8ˆ8, thus, its size is very large.

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 i
?strsplit

res = unlist(strsplit(lorem, "(?<=\\.)\\s(?=[A-Z])", perl=TRUE))
mixed = list(sample(res, length(res)))
sapply(mixed, paste, collapse = " ")
```

```
## [1] "Mauris at sodales augue.  Cras suscipit id nibh lacinia elementum. Donec vehicula sagittis nisi
```

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.

```
split_str = function(x) {
  strsplit(x, split = ", ")[[1]]
}

male_boomer = split_str("Theodore, Bernard, Gene, Herbert, Ray, Tom, Lee, Alfred, Leroy, Eddie")
female_boomer = split_str("Gloria, Joan, Dorothy, Shirley, Betty, Dianne, Kay, Marjorie, Lorraine, Mild

male_genx = split_str("Marc, Jamie, Greg, Darryl, Tim, Dean, Jon, Chris, Troy, Jeff")
female_genx = split_str("Tracy, Dawn, Tina, Tammy, Melinda, Tamara, Tracey, Colleen, Sherri, Heidi")

male_millennial = split_str("Zachary, Dylan, Christian, Wesley, Seth, Austin, Gabriel, Evan, Casey, Lui
female_millennial = split_str("Samantha, Alexis, Brittany, Lauren, Taylor, Bethany, Latoya, Candice, Br


boomer = list("M"=male_boomer, "F"=female_boomer)
genx = list("M"=male_genx, "F"=female_genx)
millennial = list("M"=male_millennial, "F"=female_millennial)

data = list("Boomer"=boomer, "GenX"=genx, "Millennial"=millennial)
```

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, Millennial. 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 = rep(NA,n),
#
#TO-DO
```

Packages

Install the package `pacman` using regular base R.

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

```
## Installing package into '/home/rstudio-user/R/x86_64-pc-linux-gnu-library/4.0'
## (as 'lib' is unspecified)
```

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 : 100)
```

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:

```
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] # reverse index
  }

  v_rev
}

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.

```
#' Nearest Neighbor Classifier
#'
#' Classify an observation based on the label of the closest observation
#' in the set of training observations
#'
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    Vector of training data labels
#' @param d           A distance function that takes in two row vectors
```

```r
#' @param xtest       A test observation as a row vector
#' @return            Predicted label for Xtest
nn_algorithm_predict = function(Xinput, y_binary, xtest, d=function(v1, v2) {sum((v1 - v2)^2)}){
  n = nrow(Xinput)
  distances = array(NA, n)  # all distances

  for (i in 1:n) {
    distances[i] = d(Xinput[i,], xtest) # get diff between row vectors
  }

  y_binary[which.min(distances)] # predicted label
}
```

Write a few tests to ensure it actually works:

```r
Xinput = as.matrix(cbind(c(1, 1, 2, 3, 3, 4), c(1, 2, 1, 3, 4, 3)))
Xtest = as.matrix(c(1, 3, 4, 2, 1, 2))
y_bin = factor(c(0, 0, 0, 1, 1, 1))
nn_algorithm_predict(Xinput, y_bin, Xtest)
```

```
## [1] 0
## Levels: 0 1
```

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.

```r
#' Nearest Neighbor Classifier
#'
#' Classify an observation based on the label of the closest observation
#' in the set of training observations
#'
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    Vector of training data labels
#' @param d           A distance function that takes in two row vectors
#' @param xtest       A test observation as a row vector
#' @return            Predicted label for Xtest
nn_algorithm_predict = function(Xinput, y_binary, xtest, d=function(v1, v2) {sum((v1 - v2)^2)}){
  n = nrow(Xinput)
  distances = array(NA, n)  # all distances

  for (i in 1:n) {
    distances[i] = d(Xinput[i,], xtest) # get diff between row vectors
  }

  y_binary[which.min(distances)] # predicted label
}
```

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.

```r
#' Nearest Neighbor Classifier
#'
```

```r
#' Classify an observation based on the label of the closest observation
#' in the set of training observations
#'
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    Vector of training data labels
#' @param k           Scalar for the number of closest observations we want
#' @param d           A distance function that takes in two row vectors
#' @param xtest       A test observation as a row vector
#' @return            Predicted label for Xtest
nn_algorithm_predict = function(Xinput, y_binary, xtest, k=NULL, d=function(v1, v2) {sum((v1 - v2)^2)})
  n = nrow(Xinput)
  distances = array(NA, n)  # all distances
  if (is.null(k)) {
    k = sqrt(n)
  }  # check that k is not null

  for (i in 1:n) {
    distances[i] = d(Xinput[i,], xtest) # get diff between row vectors
  }
  # get the
  k_nearest = apply(distances, 1, order)[1:k] # indices of the k nearest
  k_nearest_labels = y_binary[k_nearest] # labels of the k nearest

  y_hat = NULL
  # check if there is a tie

  if (sum(as.numeric(k_nearest_labels)) == length(k_nearest_labels)) {
    y_hat = sample(0:1, 1) # choose 0 or 1 randomly
  } else {
    y_hat = mode(k_nearest_labels)  # if no tie, get the mode
  }

  y_hat
}

nn_algorithm_predict(Xinput, y_bin, Xtest, 5)
```

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

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.

```r
data(iris)
pacman::p_load(skimr)
# skim(iris) IF I UNCOMMENT THIS LINE I CANT COMPILE TO PDF
```

The iris dataset contains 4 features. There are 150 observations. Each observation is labeled as virginica, setosa, or versicolor. We can easily see the mean value of the length and width for both sepal and petal.

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.integer(iris$Species == "setosa")
y
```

```
##   [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 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 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
##  [75] 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
```

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

```
mode = function(x) {
  names(sort(table(x), decreasing = TRUE)[1])
}
```

- 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`.

```
X = as.matrix(iris$Sepal.Length)

n = nrow(X)
num_errors_by_param = matrix(NA, nrow=n, ncol=2)  # init matrix
colnames(num_errors_by_param) = c("threshold_param", "num_errors") # add cols

for (i in 1:n) {
  threshold = X[i,]
  num_errs = sum((X > threshold) != y)
  num_errors_by_param[i, ] = c(threshold, num_errs)
}

best_row = order(num_errors_by_param[, "num_errors"])[1]
threshold = c(num_errors_by_param[best_row, "threshold_param"], use.names=FALSE)

threshold
```

```
## [1] 7
```

What is the total number of errors this model makes?

```
total_num_errs = sum(num_errors_by_param[, 2])
total_num_errs
```

```
## [1] 7204
```

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

```
threshold
```

```
## [1] 7
```

```
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
```

The model doesnt make sense because the max is already value if 7, so it would only predict one class.

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:

```
#' Perceptron with p features
#'
#' Implementation of the Perceptron model with p features.
#'
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    Vector of training data labels
#' @param MAX_ITER    Number of iterations the models must do to converge
#' @param w           Weight vector used to learn the parameters
#'
#' @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) {
  p = ncol(Xinput) # get the number of features
  if (is.null(w)) {
    # check if w is NULL
    w = rep(0, p + 1) # p + 1 since we include the bias
  } else if (length(w) != p + 1) {
    stop("The length of w must be the nrow(Xinput) + 1")
  }

  X_with_bias = as.matrix(cbind(1, Xinput))
  n = nrow(X_with_bias)

  for (iter in 1 : MAX_ITER) {
    for (i in 1 : n) {
      x_i = X_with_bias[i, ] # get the ith observation
      yhat = ifelse(sum(x_i * w) > 0, 1, 0)  # predicted label
      y_i = y_binary[i] # true label
      # update weights and biases
      for (j  in 1:ncol(X_with_bias)) {
        w[j] = w[j] + (y_i - yhat) * x_i[j]
      }
    }
  }

  w # return weight vector
}
```

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
)
```

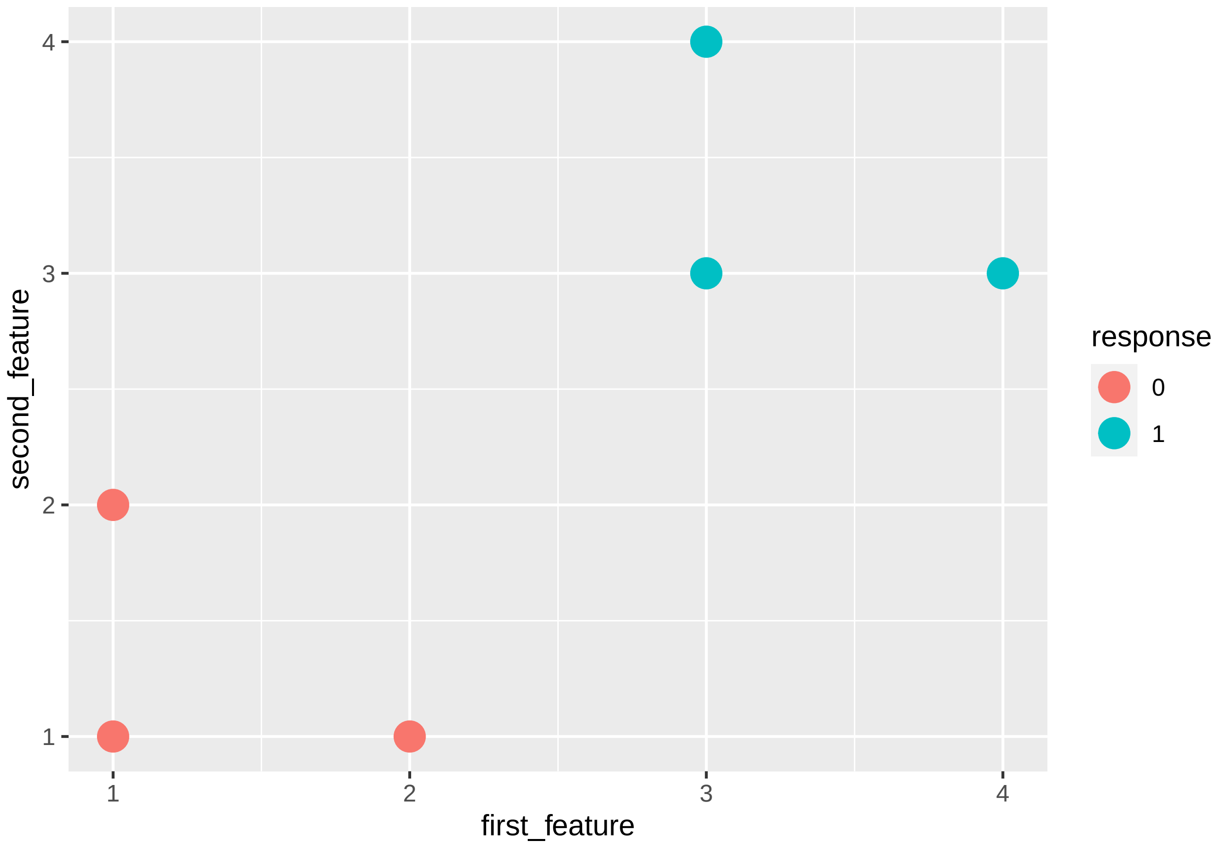
We haven't spoken about visualization yet, but it is important we do some of it now. Thus, I will write this code for you and you will just run it. First we load the visualization library we're going to use:

```
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 can see in the picture that we have two different classes. The data is linearly separable because we can easily draw a line that separates both classes.

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

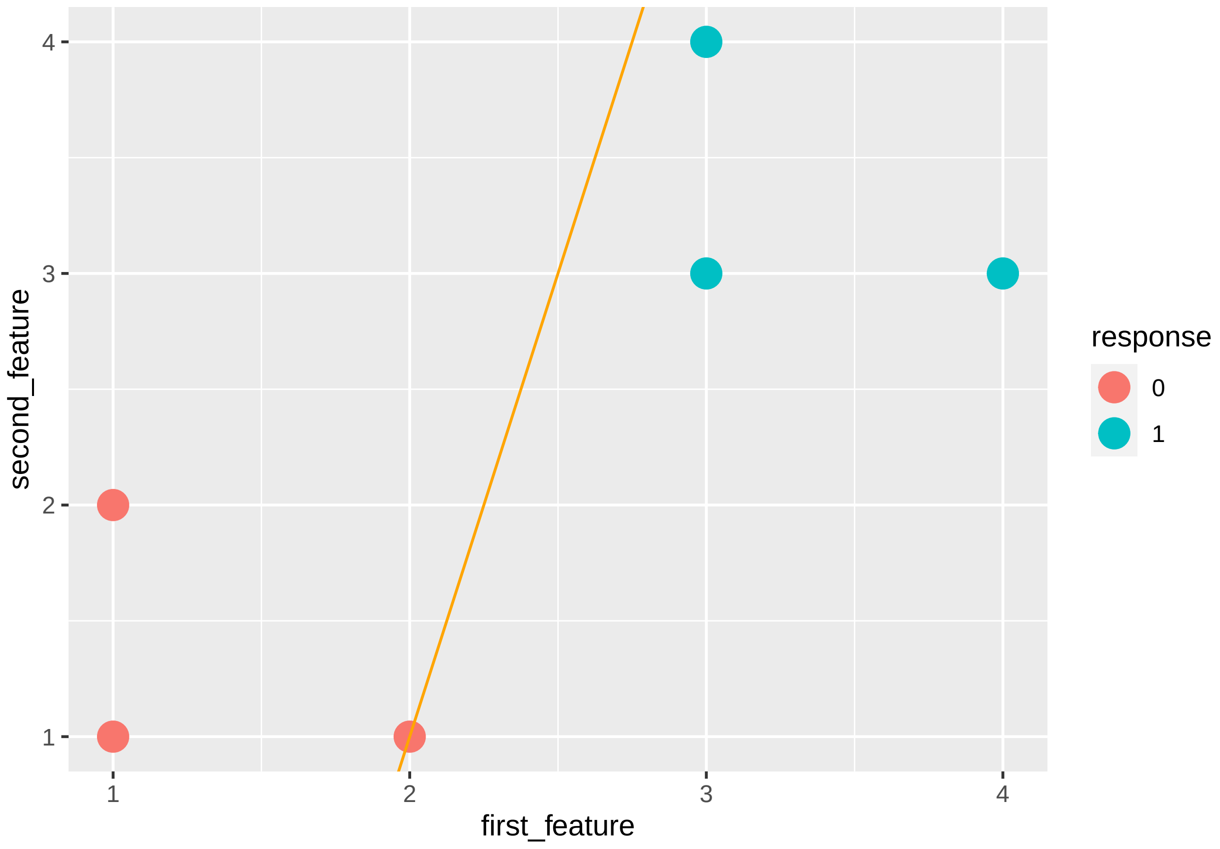
```
w_vec_simple_per = perceptron_learning_algorithm(
  cbind(Xy_simple$first_feature, Xy_simple$second_feature),
  as.numeric(Xy_simple$response == 1))
w_vec_simple_per
```

```
## [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 intercept is -7 and the slope is 4. This is the line the perceptron was able to create. It somewhat seperates the data but could be better.

```
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?

The line is not satisfying because we can easily create a line that will maximize the distance between each of the classes. The line does a mediocre job at separating the data.

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
```