Lab 2

Hubert Majewski

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) {
        ret <- vector(length = length(v))

        for(i in 1:length(v)) {
          ret[i] <- v[length(v) - i + 1]
        }

        return(ret)
}
head(v)
```

```
## [1] 1 2 3 4 5 6
```

```
tail(my_reverse(v))
```

```
## [1] 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(matrix, dim_to_rev = NULL) {

    #Determine weather to do rows or cols
    if (is.null(dim_to_rev))
        dim_to_rev <- nrow(matrix) >= ncol(matrix) ? "Row" : "Column"

    #reverse
    if (dim_to_rev == "Row")
        matrix[my_reverse(1 : nrow(matrix)), ]
  else if (dim_to_rev == "Column")
```

```
    matrix[my_reverse(1 : ncol(matrix))]
    else #Default: Do a flip if the argument is invalid
        return(flip_matrix(matrix, NULL))
}
```

- 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();
c <- 8;

for (i in 1:c) {
    my_list[[LETTERS[i]]] = array(data = 1:i^i, dim = 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
```

```
?object.size
```

```
## starting httpd help server ... done
```

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?

To quote: "Provides an estimate of the memory that is being used to store an R object." This does make sense because our matrix dimensions are exponential so the number of bytes used is also exponential.

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

```
remove(list=ls())

#Redefine reverse as its used later
my_reverse = function(v) {
        ret <- vector(length = length(v))

        for(i in 1:length(v)) {
          ret[i] <- v[length(v) - i + 1]
        }

        return(ret)
}
```

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 

split <- strsplit(lorem, split = "\\.\\ ")

split <- unlist(split)

sample(split)
```

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

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.

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

#Splitstring for all
M_B <- strsplit(M_B, split = ",")
M_G <- strsplit(M_G, split = ",")
M_M <- strsplit(M_M, split = ",")
F_B <- strsplit(F_B, split = ",")
F_G <- strsplit(F_G, split = ",")
F_M <- strsplit(F_M, split = ",")

#Build
M <- list(Boomer = M_B, GenX =M_G, Millenial=M_M)
F <- list(Boomer = F_B, GenX =F_G, Millenial=F_M)

names <- list(Boomer=M_B, GenX=M_G, Millenial=M_M, Boomer=F_B, GenX=F_G, Millenial=F_M)

#Final
datalist <- list(Male = M, Female = F)
datalist
```

```
## $Male
## $Male$Boomer
## $Male$Boomer[[1]]
##  [1] "Theodore" " Bernard" " Gene"    " Herbert" " Ray"     " Tom"
##  [7] " Lee"     " Alfred"  " Leroy"   " Eddie"
##
##
## $Male$GenX
## $Male$GenX[[1]]
##  [1] "Marc"    " Jamie"  " Greg"   " Darryl" " Tim"    " Dean"   " Jon"
##  [8] " Chris"  " Troy"   " Jeff"
##
##
## $Male$Millenial
## $Male$Millenial[[1]]
##  [1] "Zachary"    " Dylan"     " Christian" " Wesley"    " Seth"
##  [6] " Austin"    " Gabriel"   " Evan"      " Casey"     " Luis"
##
##
##
## $Female
## $Female$Boomer
## $Female$Boomer[[1]]
##  [1] "Gloria"    " Joan"     " Dorothy"  " Shirley"  " Betty"    " Dianne"
##  [7] " Kay"      " Marjorie" " Lorraine" " Mildred"
##
##
## $Female$GenX
## $Female$GenX[[1]]
##  [1] "Tracy"    " Dawn"    " Tina"    " Tammy"   " Melinda" " Tamara"
##  [7] " Tracey"  " Colleen" " Sherri"  " Heidi"
##
##
## $Female$Millenial
## $Female$Millenial[[1]]
##  [1] "Samantha"  " Alexis"   " Brittany" " Lauren"   " Taylor"   " Bethany"
##  [7] " Latoya"   " Candice"  " Brittney" " Cheyenne"
```

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

#Setup
treatment <- rep(letters[1:14], each=n/14)
variation <- rep(c("A", "B", "C"), each=n/14/3)
gender <- rep(c("M", "F"), each=n/14/3/2)
gen <- rep(c("Boomer", "Genx", "Millenial"), each=n/14/3/2/3)
names <- rep(unlist(datalist), times=14*3)

#Special case for Y as No Answer
Y <- rep(NA, n)

#Add everything into a frame
X <- data.frame(treatment, variation,gender, gen, names, Y)

head(X)
```

```
##   treatment variation gender    gen    names  Y
## 1         a         A      M Boomer Theodore NA
## 2         a         A      M Boomer  Bernard NA
## 3         a         A      M Boomer     Gene NA
## 4         a         A      M Boomer  Herbert NA
## 5         a         A      M Boomer      Ray NA
## 6         a         A      M Boomer      Tom NA
```

```
summary(X)
```

```
##   treatment          variation            gender              gen
##  Length:2520        Length:2520        Length:2520        Length:2520
##  Class :character   Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character   Mode  :character
##     names               Y
##  Length:2520        Mode:logical
##  Class :character   NA's:2520
##  Mode  :character
```

Packages

Install the package `pacman` using regular base R.

```
if ("pacman" %in% rownames(installed.packages()) == FALSE)
  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)
tryCatch(expect_equal(v, -100 : 101), error = function(cond) return(paste("Failed Successfully: ", cond
```

```
## [1] "Failed Successfully:  Error: `v` not equal to -100:101.\nLengths differ: 201 is not 202\n"
```

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:

```
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 Prediction Classifier
#'
#' This method seeks to pridict the closest label given the observations and associated labels.
#'
#' @param Xinput      All of the features that will be used to predict the label which are measured aga
#' @param y_binary    The labels that will be used for testing
#' @param Xtest       Set of test parameters to test against the model
#' @return            Returns a probable lablel given the test and the model
nn_algorithm_predict = function(Xinput, y_binary, Xtest){

    dis <- array(NA, nrow(Xinput))
```

```r
    for(i in 1:n)
        dis[i] <- sum((xinput[i, ]-xtest)^2)

    return(y_binary[which.min(dis)])

}
```

Write a few tests to ensure it actually works:

```r
#Import data Iris
data(iris)

#Select label column into y
yinput <- iris$Species

#Everything but the lables
tempIris <- iris
tempIris$Species <- NULL
xinput <- tempIris


xtest <- c(5.87, 1.87, 420.69, 3)

nn_algorithm_predict(xinput, yinput, xtest)
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

```r
xtest <- c(0.98, 2.5, 1, 8)

nn_algorithm_predict(xinput, yinput, xtest)
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

```r
xtest <- c(4.7, 12, 4, 23)

nn_algorithm_predict(xinput, yinput, xtest)
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

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 Prediction Classifier
#'
#' This method seeks to pridict the closest label given the observations and associated labels.
#'
```

```r
#' @param Xinput      All of the features that will be used to predict the label which are measured aga
#' @param y_binary    The labels that will be used for testing
#' @param Xtest       Set of test parameters to test against the model
#' @param d           A function that will compute the distance between row vectors. Default: Euclidean
#' @return            Returns a probable lablel given the test and the model
nn_algorithm_predict = function(Xinput, y_binary, Xtest, d = function(x,y) {return(sum((x[i, ]-y)^2))})

    dis <- array(NA, nrow(Xinput))

    for(i in 1:n)
        dis[i] <- d(xinput, xtest)

    return(y_binary[which.min(dis)])

}
```

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.

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)
```

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	

We see from the output above that the dataset attempts to classify three different flower species. The dataset attempts to classify using the petals length and width and the sepal length and width.

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.

```
data(iris)
iris <- iris[iris$Species!="virginica", ]
head(iris)
```

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

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=="versicolor")
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
##  [38] 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 1 1
##  [75] 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
```

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

```
mode <- function(x) {
    return(as.numeric(
        names(
            sort(
                table(x)
            )[1]
        )
    ))
}

mode(y)
```

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

- 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`.

```
err <- matrix(NA, nrow=nrow(iris), ncol=2)
colnames(err) <- c("Threshold", "Count")

for(i in 1:nrow(iris)) {

    #Get threshold
    threshold <- iris$Sepal.Length[i]

    #Count all that surpass
    count <- sum(y != (iris$Sepal.Length > threshold))
    err[i, ] <- c(threshold, count)
}

threshold <- c(err[(order(err[, "Count"])[1]), "Threshold"])
threshold
```

```
## Threshold
##       5.4
```

What is the total number of errors this model makes?

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

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

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

```
threshold
```

```
## Threshold
##       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
```

This models performance has sense. The summaries show the predicted length of 5.4. The length is the mean of both the verticolor and the setosa. In otherwords, the median of 5 and 5.9 is about 5.4

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

```
g = function(x){
    #threshold consists of the sepal.length feature
    return(as.numeric(x > threshold))
}
```

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 Linear Algorithm
#'
#' Computes a line of a linearly separable dataset
#'
#' @param Xinput      Input matrix in the form of n by p. It consists of the training data.
#' @param y_binary    Separable features associated with every tuple of the data.
#' @param MAX_ITER    Number of iterations for the approximation for
#' @param w           Initialization of the vector of dimensions p + 1
#'
#' @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){

    #Add 1 column to matrix making P + 1 size
  Xinput = as.matrix(cbind(1,Xinput))

  z <- rep(0, ncol(Xinput))

  for (i in 1 : MAX_ITER){
    for (j in 1 : nrow(Xinput)) {

        #Get tuple
      x <- Xinput[j, ]

      #Compute y hat
      yHAT <- if(sum(x * z) > 0) 1 else 0

      #Generate new weights
      for(k in 1:ncol(Xinput)){
        z[k] <- z[k] + (y_binary[j] - yHAT) * x[k]
      }
    }
  }

  return(z)
}
```

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 $\mathbb{D}$.

```r
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
)
```

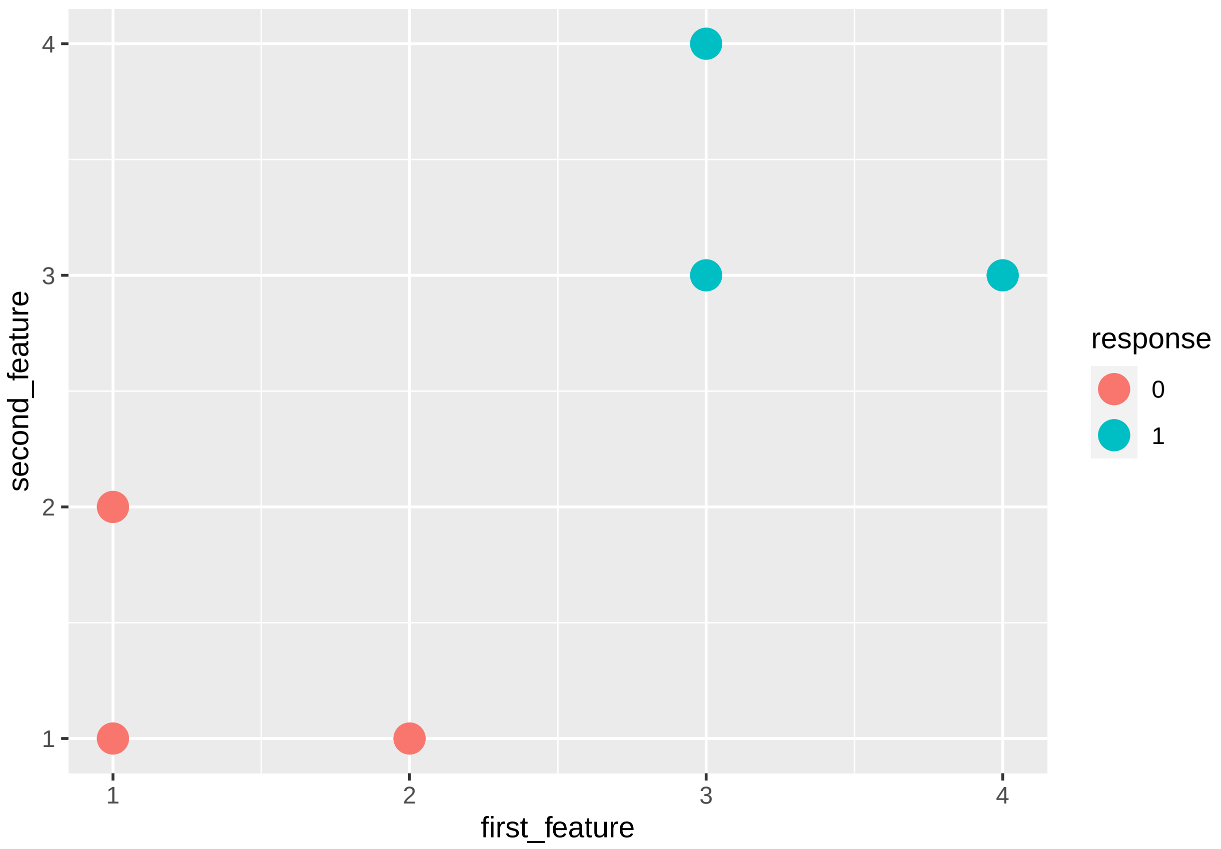
We haven't spoken about visualization yet, but it is important we do some of it now. Thus, I will write this code for you and you will just run it. First we load the visualization library we're going to use:

```r
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.

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



The chart above represents our two grouped plotted points. The red represents the first_feature while the blue represents the second_feature in the dataset.

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

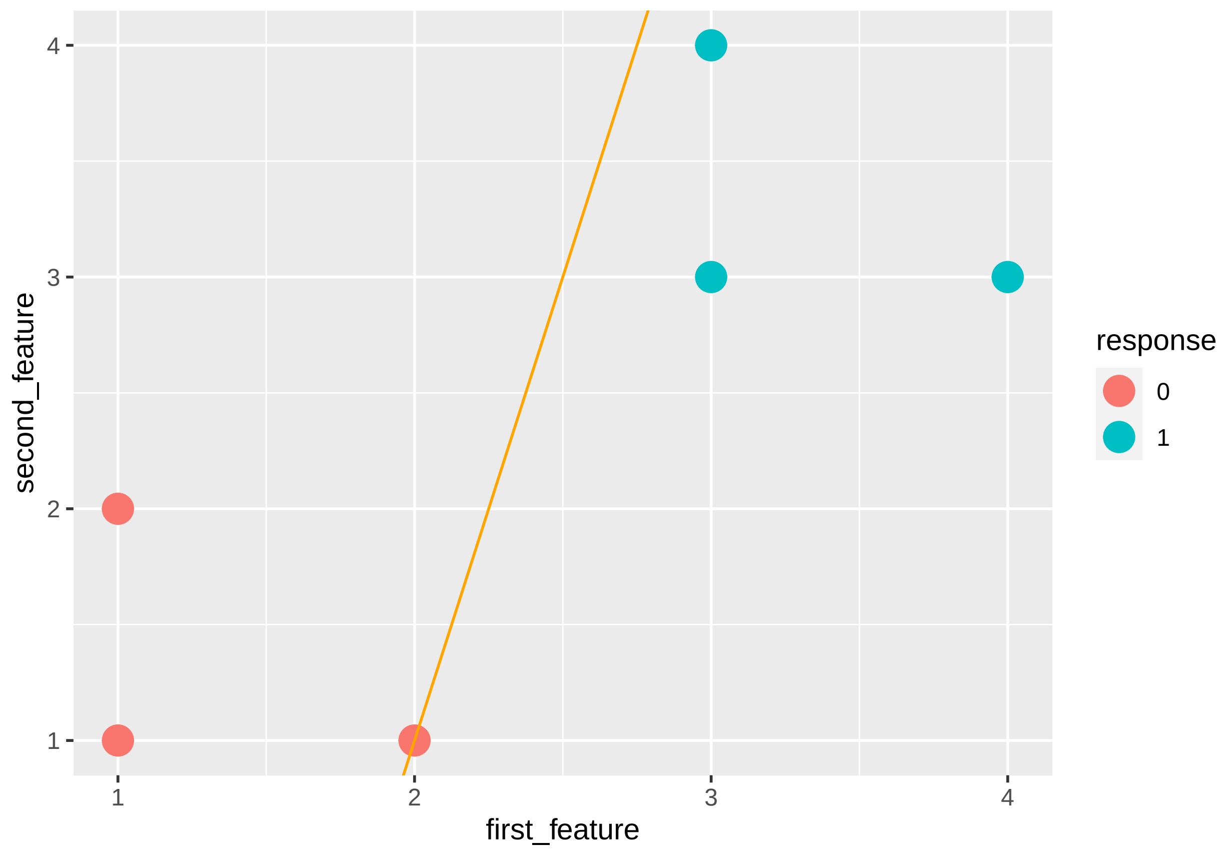
```
w_vec_simple_per = perceptron_learning_algorithm(
  cbind(Xy_simple$first_feature, Xy_simple$second_feature),
  as.numeric(Xy_simple$response == 1))

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 result is the representation of the line y = mx + b. 4 from the result is the slope (m). The y intercept is -1.

```
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
```



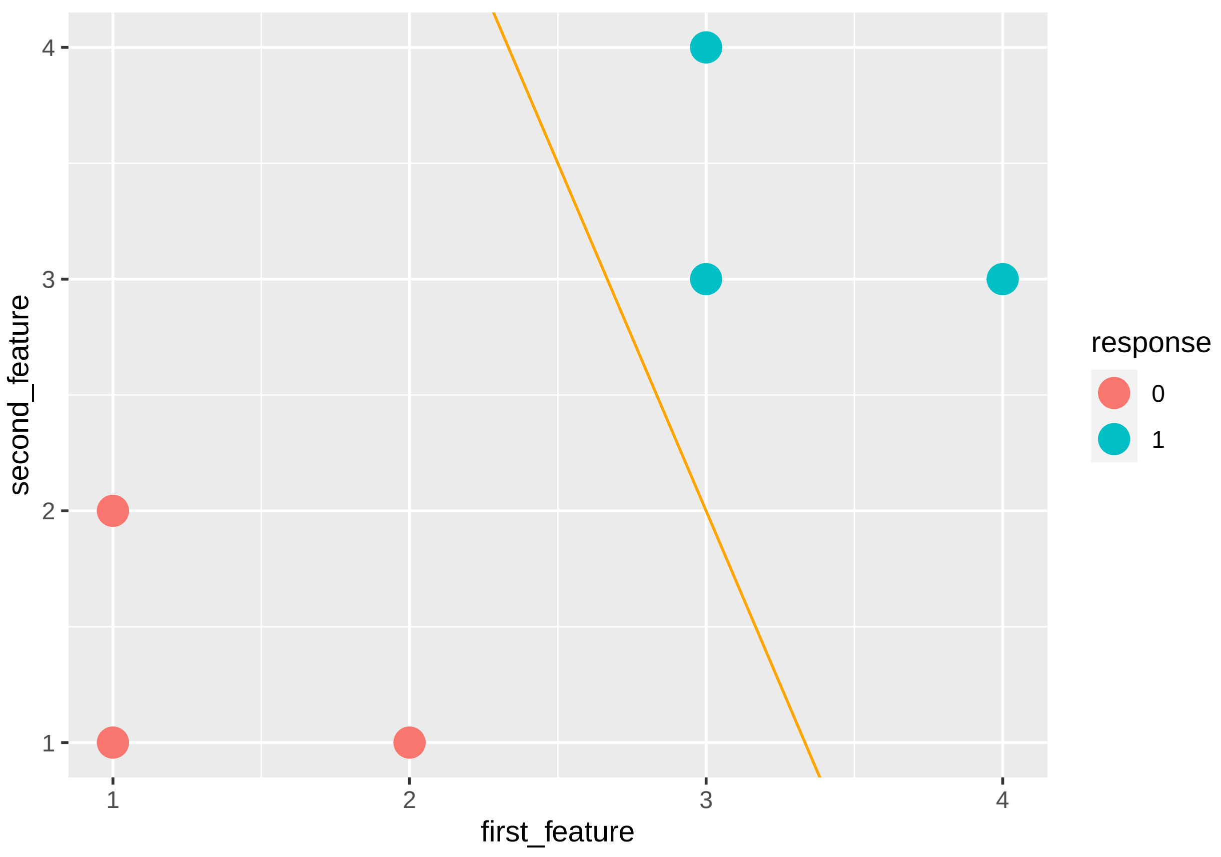
Explain this picture. Why is this line of separation not "satisfying" to you?

While it does technically split both datasets, it is not the most optimal solution. We can clearly see that the more optimal solution would equally divide between That is why it is not satisfying to me. See below for a better correction where the line does not look like its intersecting a point

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.

```r
#' Perceptron Linear Algorithm
#'
#' Computes a line of a linearly separable dataset
#'
#' @param Xinput      Input matrix in the form of n by p. It consists of the training data.
#' @param y_binary    Separable features associated with every tuple of the data.
#' @param MAX_ITER    Number of iterations for the approximation for
#' @param w           Initialization of the vector of dimensions p + 1
#'
#' @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){

    #Add 1 column to matrix making P + 1 size
  Xinput = as.matrix(cbind(1,Xinput))

  z <- rep(0, ncol(Xinput))

  for (i in 1 : MAX_ITER){
    for (j in 1 : nrow(Xinput)) {

        #Get tuple
      x <- Xinput[j, ]

      #Compute y hat
      yHAT <- if(sum(x * z) >= 0) 1 else 0

      #Generate new weights
      for(k in 1:ncol(Xinput)){
        z[k] <- z[k] + (y_binary[j] - yHAT) * x[k]
      }
    }
  }

  return(z)
}
```

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

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
```



We can see there is a more better slobe in accordance with the dataset.