---

title: "Lab 2"

author: "Asher Katz"

output: pdf_document

date: "11:59PM February 24, 2022"

More Basic R Skills

* Create a function `my_reverse` which takes as required input a vector `v` 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.

```{r}
#TO-DO
v = 1:100
#a vector
my_reverse = function(v){
  n = length(v)
  vr = array(data = NA, dim = n)
  # an array of NA's with n spaces, so it can fit v
  for (i in 1:n){
    vr[i] = v[n - i + 1]
    # vr[i] = the reversed v
  }

  vr
}
head(v)
tail(my_reverse(v))
```

* 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. If the number of rows is equal to the number of columns, throw an error.

```{r}
#TO-DO
flip_matrix = function(M, dim_to_rev = NA){
  nr = nrow(M)
  nc = ncol(M)
```

```
  if( is.na(dim_to_rev) ) {
    if( nr == nc ) { # because "the number of rows is equal to the number of columns, throw an error"
      stop( "unspecified dem_to_rev and equals rows and columns")
    }

    if( nr > nc ){
    dim_to_rev = "r"
    }
    else{
      dim_to_rev = "c"
    }
  }


  if(dim_to_rev == "r") {
    M[nr:1,] # M = [M_nr, M_nr-1, M_nr-2, M_nr-3,...,M_1]
    }
  else if (dim_to_rev == "c") {
    M[,nc:1]
  }
  else {
    stop("dim_to_rev needs to be r or c")
    }
}


smat = matrix(1:150, nrow = 10, ncol = 15 )
flip_matrix(smat, "r")
```

* 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.

```{r}
#TO-DO
my_list = list()
my_list$A = array(1:1^1,dim = c(1))
my_list$B = array(1:2^2, dim = c(2, 2))
my_list$C = array(1:3^3, dim = c(3, 3, 3))
my_list$D = array(1:4^4, dim = c(4,4,4,4))
my_list$E = array(1:5^5, dim = c(5,5,5,5,5))
my_list$"F" = array(1:6^6, dim = c(6,6,6,6,6,6))
```

```
my_list$G = array(1:7^7, dim = c(7,7,7,7,7,7,7))

my_list$H = array(1:8^8, dim = c(8,8,8,8,8,8,8,8))

# for (i in 1:8){

#   my_list[[LETTERS[i]]] = array( data = c(1:(i^i)), dim = c(1:i))



#for letters i = A through i = H, we get a list of arrays, each sized as (i-7)x(i-6)x(i-5)x(i-4)x(i-3)x(i-2)x(i-1)x(i) and populated by values 1: i!

#   # [H: 8x8x8x8 x 8x8x8x8 ]

#   # 8^8

# }

View(my_list)

#my_list
```

Run the following code:

```{r}
?object.size

lapply(my_list, object.size)



sizez = lapply(my_list, object.size)

next_divided_by_current = c()

prev=1

cnt=0

for (i in as.numeric(sizez)){

  next_divided_by_current[cnt] = i/(prev)

  cnt = cnt + 1

  prev = i

}

next_divided_by_current[8]=NA

comparison = cbind(sizez, next_divided_by_current )

comparison
```

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?

Yes, they do make sense, exponential growth with each letter increase

#TO-DO

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

```{r}
#TO-DO
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.

```{r}
lorem = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi posuere varius volutpat. Morbi faucibus ligula id massa ultricies viverra. Donec vehicula sagittis nisi non semper. Donec at tempor erat. Integer dapibus mi lectus, eu posuere arcu ultricies in. Cras suscipit id nibh lacinia elementum. Curabitur est augue, congue eget quam in, scelerisque semper magna. Aenean nulla ante, iaculis sed vehicula ac, finibus vel arcu. Mauris at sodales augue. "

strsplit(x = lorem, split = "\\.\\s")[[1]]
```

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}
#HINT:
```

```
#strsplit("Theodore, Bernard, Gene, Herbert, Ray, Tom, Lee, Alfred, Leroy, Eddie", split = ", ")[[1]]

#TO-DO

intel_list = list()

intel_list$m = list()

intel_list$f = list()

intel_list$m$Boomer = strsplit( "Theodore, Bernard, Gene, Herbert, Ray, Tom, Lee, Alfred, Leroy, Eddie",split = ", ")[[1]]

intel_list$m$GenX = strsplit("Marc, Jamie, Greg, Darryl, Tim, Dean, Jon, Chris, Troy, Jeff",split = ", ")[[1]]

intel_list$m$Millenial = strsplit("Zachary, Dylan, Christian, Wesley, Seth, Austin, Gabriel, Evan, Casey, Luis",split = ", ")[[1]]

intel_list$f$Boomer = strsplit("Gloria, Joan, Dorothy, Shirley, Betty, Dianne, Kay, Marjorie, Lorraine, Mildred",split = ", ")[[1]]

intel_list$f$GenX = strsplit("Tracy, Dawn, Tina, Tammy, Melinda, Tamara, Tracey, Colleen, Sherri, Heidi",split = ", ")[[1]]

intel_list$f$Millenial = strsplit("Samantha, Alexis, Brittany, Lauren, Taylor, Bethany, Latoya, Candice, Brittney, Cheyenne"

,split = ", ")[[1]]

intel_list
```

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. Hint, we've been using the `rep` function using the `times` argument. Look at the `each` argument using `?rep`.

```{r}
#TO-DO

tr = c(LETTERS[1:14])

var = c("a", "b", "c")

gndr = c("m", "f")

gen = c("Millenial", "GenX", "Boomer")

mb = sample(intel_list$m$Boomer, 6)

mx = sample(intel_list$m$GenX, 6)

mm = sample(intel_list$m$Millenial, 6)

fb = sample(intel_list$f$Boomer, 6)

fx = sample(intel_list$f$GenX, 6)

fm = sample(intel_list$f$Millenial, 6)

name = c(mb,mx,mm,fb,fx,fm)

#Create a data frame with columns...
```

```
X = data.frame(



#...treatment...

  treatment = rep( tr, each = 6*3*2*3),

  #Replicate each of the 14 levels of tr 108 times, so 14*108=1512 treatments



#...variation...

  variation = rep( rep( var, each = 6*3*2*3, times = 14)),

  #Replicate each variations in var 108 times, so 108 a's, 108 b's...324 elements, then repeat that 14 times, so 14*324= 4536 variations AKA 3*1512



#...gender...

  gender = rep(rep( gndr, each = 18 ), times = 3*14),

  # rep(gndr) has 18 male + 18 female, and replicate those 36 elements 42 times, so 36*42= 1512 people



#...generation...

  generations = rep(rep(gen, each = 6, times = 2*3*14)),

  # Replicate each element of the 3 elements in gen 6 times, so 18 elements and repeat that 84 times. 18*84= 1512 generations



#...name...

  names = rep(name, 42),

  # Given a Treatment, Variation, Gender, and Generation, pick 6 unique names.



#...and y that will store all the unique unit information in this experiment.

  y = NA

)

View(X)



```


Packages

Install the package `pacman` using regular base R.


```{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.

```{r}
pacman::p_load(testthat)
```

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

```{r}
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 using the following code:

```{r}
expect_equal(my_reverse(v), rev(v))
expect_equal(my_reverse(c("A", "B", "C")), c("C", "B", "A"))
```

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 in English.

```{r}
#TO-DO
rm(list = ls())
data(iris) #load the iris dataset (as a data frame). This dataset is included in the package "datasets" which is autoloaded by default
library(skimr)
skim(iris)
```

TO-DO: describe this data

The data contains measurements of the sepals (the leaves which form the shell of the bud) and petals (the leaves which are contained in the bud) of 150 iris flowers and seeks to identify the species of each flower based on these measurements. We have 3 species, "set" AKA setosa, "ver" AKA versicolor, and "vir" AKA virginica. Species

is considered a non-numeric factor column and the measurements are numeric columns. The average of sepal length is 5.84cm, of sepal width is 3.06cm, of petal length is 3.76cm and of petal width is 1.20cm.

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.

```{r}
#TO-DO
setVir = data.frame(iris[1:100,])
View(setVir)
```

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".

```{r}
#TO-DO
y = c()
n = length(setVir[,5])

for (i in 1:n ){
  if (setVir$Species[[i]] == "setosa"){
    y[i] = 0
  }
  else{
    y[i] = 1
  }
}
y
```

* Write a function `mode` returning the sample mode of a vector of numeric values. Try not to look in the class notes.

```{r}
#TO-DO

v = c(sample(1:10, 100, replace=TRUE) )

Mode = function(x) { # mode is the number which occurs most frequently
  ux = unique(x)
```

```
  ux[ which.max( tabulate( match(x, ux))) ] # match(x,y, ...etc) = the positions an element in "y" where it first occurs in of "x"

} # tabulate counts the number of occurrences of an element in the argument, and returns it in the position (!!in the tabulate vector) where it first occurs in the argument

Mode(v)

```


* 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`.


```{r}

#TO-DO

setVir = data.frame(iris[1:100,])

# what length of sepal signifies set

setVir$y = y

SL = setVir$Sepal.Length

y_logical = setVir$y



num_errors_by_parameter = matrix(NA, nrow = n, ncol = 2) #a 2 by n matrix which records the number of errors of every given Sepal length

colnames(num_errors_by_parameter) = c("threshold_param", "num_errors")



for (i in 1:n){

  threshold = SL[i]

  num_errors = sum(as.numeric(SL>threshold) != y_logical) # (SL > threshold) creates a binary list of less than, not less than. An error is when that list does not match somewhere with y_binary. the sum of those errors is num_errors

  num_errors_by_parameter[i,] = c(threshold, num_errors) #our 2 by n errors matrix is being filled with (threshold,num_errors) pairs



}



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

#now grab the smallest num errors

best_row = order(num_errors_by_parameter[, "num_errors"])[1]



x_star = c(num_errors_by_parameter[best_row, "threshold_param"], use.names = FALSE)

x_star







```


What is the total number of errors this model makes?

2796 or 11

```{r}
#TO-DO
sum(num_errors_by_parameter[,2])
```

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

```{r}
threshold
summary(iris[iris$Species == "setosa", "Sepal.Length"])
summary(iris[iris$Species == "versicolor", "Sepal.Length"])
```

TO-DO: Write your answer here in English.

Yes, I think it does make sense given the summary. It fits well within Setosa's values and is well below most of the Versicolor's values

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

```{r}
#TO-DO

g = function(x){
if (x > x_star){
  return(" Versicolor")
}
  else{
    return("Setosa")
  }


}
g(5.9)
g(1)
```

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:

```{r}
#' TO-DO: Explain what this function does in a few sentences
#'
#' @param Xinput      TO-DO: this is our features which are input into the perceptron
#' @param y_binary    TO-DO: This is our our output, 0 being no, 1 being yes
#' @param MAX_ITER    TO-DO: This is the number of attempts we will make to get the best threshold line
#' @param w           TO-DO: this is the weight vector representing our best threshold line. Every iteration should decrease the error a bit more.
#'
#' @return            The computed final parameter (weight) as a vector of length p + 1
  #TO-DO
#
# perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w = NULL){
#   if(is.null(w)){
#     w = rep(0, ncol(Xinput)+1)
#   }
#
#   n = nrow(Xinput)
#
#   Xinput = cbind(1,Xinput)
#   for(iter in 1:MAX_ITER){
#     y_hat_vec = as.numeric(Xinput%*%w > 0) # y_hat is 1s and 0s
#     w_0 = w
#     for (i in 1:n){
#       w = w_0 + (y_binary[i] - y_hat_vec[i])*Xinput[i,]
#     }
#     if (all(w == w_0)){
#       break
#
#     }
#   }
#   w
#
# }
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w = NULL){
```

```
  Xinput = as.matrix(cbind(1, Xinput))
  n = nrow(Xinput)
  p = ncol(Xinput)
  w = rep(0, p)
  for (i in 1 : MAX_ITER) {
    for (j in 1 : n) {
      x_j = Xinput[j,]
      y_hat = ifelse(sum(x_j * w) > 0, 1, 0)
      for (k in 1 : p) {
        w[k] = w[k] + (y_binary[j] - y_hat) * x_j[k]
      }
    }
  }

  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 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
)
Xy_simple
```

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
```

TO-DO: Explain this picture.

Each point is (first feature, second feature) pair, and if its value is 1, its colored blue, if 0, red

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

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

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.

w1 = w0x + b   -1 = 4x -7, b is the bias, w0, w1 are weights.

The intercept is -7 and the slope is 4.

TO-DO

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

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

because it barely avoids the no at (2,1), when it can easily avoid it

TO-DO

<!-- 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} -->

<!-- #TO-DO -->

<!-- ``` -->

Support Vector Machine vs. Perceptron

We recreate the data from the previous lab and visualize it:

```{r}
pacman::p_load(ggplot2)
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
)
simple_viz_obj = ggplot(Xy_simple, aes(x = first_feature, y = second_feature, color = response)) +
  geom_point(size = 5)
simple_viz_obj
```

Use the `e1071` package to fit an SVM model to the simple data. Use a formula to create the model, pass in the data frame, set kernel to be `linear` for the linear SVM and don't scale the covariates. Call the model object `svm_model`. Otherwise the remaining code won't work.

```{r}
pacman::p_load(e1071)
svm_model = svm(
  formula = (y_i - 0.5)((w_vec%*%x_i_vec) - b) >= 0.5,
  data = Xy_simple$response,
  x = as.matrix(Xy_simple[, 2 : 3]), # can't be a vector here
  kernel = "linear",
  scale = FALSE
)
 as.matrix(Xy_simple[, 2 : 3])

#svm(formula, data = NULL, ...{for us this is kernel = linear], subset, na.action = na.omit, scale = TRUE)
```

and then use the following code to visualize the line in purple:

```{r}
w_simple_svm = c(
  svm_model$rho, #the b term
  -t(svm_model$coefs) %*% cbind(Xy_simple$first_feature, Xy_simple$second_feature)[svm_model$index, ] # the other terms
)
simple_svm_line = geom_abline(
    intercept = -w_simple_svm[1] / w_simple_svm[3],
    slope = -w_simple_svm[2] / w_simple_svm[3],
    color = "purple")
simple_viz_obj + simple_svm_line
```

Source the `perceptron_learning_algorithm` function from lab 2. Then run the following to fit the perceptron and plot its line in orange with the SVM's line:

```{r}
w_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_simple_per[1] / w_simple_per[3],
    slope = -w_simple_per[2] / w_simple_per[3],
    color = "orange")
simple_viz_obj + simple_perceptron_line + simple_svm_line
```

Is this SVM line a better fit than the perceptron?

Yes

TO-DO

Now write pseuocode for your own implementation of the linear support vector machine algorithm using the Vapnik objective function we discussed.

Note there are differences between this spec and the perceptron learning algorithm spec in question #1. You should figure out a way to respect the `MAX_ITER` argument value.

```{r}

#' Support Vector Machine

#

#' This function implements the hinge-loss + maximum margin linear support vector machine algorithm of Vladimir Vapnik (1963).

#'

#' @param Xinput      The training data features as an n x p matrix.

#' @param y_binary    The training data responses as a vector of length n consisting of only 0's and 1's.

#' @param MAX_ITER    The maximum number of iterations the algorithm performs. Defaults to 5000.

#' @param lambda      A scalar hyperparameter trading off margin of the hyperplane versus average hinge loss.

#'                    The default value is 1.

#' @return            The computed final parameter (weight) as a vector of length p + 1

linear_svm_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 5000, lambda = 0.1){

  #TO-DO: write pseudo code in comments

# for (1:MAX_ITER){

#   initialize the w and Sum of Hinge Error variables

#   }

#

# for (1 to nrow(Xinput)){

#   set ideal w so that ||w|| satisfies (y_binary_i - 0.5)(w %*% Xinput_i - b) >=  0.5

#   min_w = min(w, min_w)

#   compute sum of the hinge errors by summing the max(0, 0.5 - (y_binary_i - 0.5)(w*Xinput_i - b))

#   optimize wedge with the hyper parameter lambda ( (1/nrow)*SHE + lambda*(||w||)^2)

#   return(w)

# }

#

#

}

```

<!-- If you are enrolled in 342W the following is extra credit but if you're enrolled in a masters section, the following is required. Write the actual code. You may want to take a look at the `optimx` package. You can feel free to define another function (a "private" function) in this chunk if you wish. R has a way to create public and private functions, but I believe you need to create a package to do that (beyond the scope of this course). -->

```{r}

#' This function implements the hinge-loss + maximum margin linear support vector machine algorithm of Vladimir Vapnik (1963).

#'

#' @param Xinput      The training data features as an n x p matrix.

#' @param y_binary    The training data responses as a vector of length n consisting of only 0's and 1's.
```

#' @param MAX_ITER   The maximum number of iterations the algorithm performs. Defaults to 5000.

#' @param lambda   A scalar hyperparameter trading off margin of the hyperplane versus average hinge loss.

#'   The default value is 1.

#' @return   The computed final parameter (weight) as a vector of length p + 1

linear_svm_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 5000, lambda = 0.1){

#TO-DO

}

```

#

If you wrote code (the extra credit), run your function using the defaults and plot it in brown vis-a-vis the previous model's line:

#

```{r}

svm_model_weights = linear_svm_learning_algorithm(X_simple_feature_matrix, y_binary)

my_svm_line = geom_abline(

intercept = svm_model_weights[1] / svm_model_weights[3],#NOTE: negative sign removed from intercept argument here

slope = -svm_model_weights[2] / svm_model_weights[3],

color = "brown")

simple_viz_obj + my_svm_line

```

#

Is this the same as what the `e1071` implementation returned? Why or why not?

#

TO-DO

```

Multinomial Classification using KNN

Write a k = 1 nearest neighbor algorithm using the Euclidean distance function. The following comments are 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.

```{r}

#' TO-DO: Provide a name for this function

#'

#' TO-DO: Explain what this function does in a few sentences

#' If we have a particular value and we want to predict it's outcome, this algorithm will look at the k most similar values which we know the outcomes for and use those as the predicted outcome of our particular value. It is using actual euclidean distance to find the k (most similar) nearest values.

#' @param Xinput      TO-DO: Our input features and accompanying data

#' @param y_binary    TO-DO: Our data outputs

#' @param Xtest       TO-DO: A value which we want to predict the outcome of, using our k nearest neighbors algorithm

#' @return            TO-DO: The predicted outcome

```
nn_algorithm_predict = function(Xinput, y_binary, Xtest){

  y_hat_vec = rep(NA, length(Xtest)) #all the predictions

  for(istar in 1:length(Xtest)){

    y_hat_vec[istar] = y_binary[which.min(rowSums(t(t(Xinput) - Xtest[istar])^2))] #euclidean distance
squared,

  }

  y_hat_vec

}
```

Write a few tests to ensure it actually works:

```{r}

#TO-DO

data(iris)

pacman::p_load(skimr)

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

y = as.integer(iris$Species == "versicolor")

iris

iris = iris[, 1:4]

x_test1 = c(5.1, 3.6, 1.4, 0.1) # 0

x_test2 = c(5.0, 3.5, 1.4, 0.3) # 0

x_test3 = c(5.9, 3.0, 4.1, 1.5) # 1

x_test4 = c(6.4, 3.2, 4.5, 1.5) # 1

nn_algorithm_predict(iris, y, x_test1) # is 0 !

nn_algorithm_predict(iris, y, x_test2) # is 0 !

nn_algorithm_predict(iris, y, x_test3) # is 1 !

nn_algorithm_predict(iris, y, x_test4) # is 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}

#' TO-DO: Provide a name for this function

#'

#' TO-DO: Explain what this function does in a few sentences

#'

#' @param Xinput      TO-DO: the matrix of features of data

#' @param y_binary    TO-DO: the vector of data outcome labels

#' @param Xtest       TO-DO: a test observation as a row vector
```

```
#' @param d           TO-DO: a distance function which take inputs to different row vectors

#' @return            TO-DO: the predicted outcome for the test

nn_algorithm_predict = function(Xinput, y_binary, Xtest){

  if (is.null(d)){

    d = function(xi, xstar){

      sum(xi - xstar)^2

    }

  }

  y_hat_vec = array(NA, nrow(Xtest)) #all the predictions

  for( istar in 1:nrow(Xtest)){

    dists - array(NA, nrow(Xinput))

    for (i in 1:nrow(Xinput))

      y_hat_vec[istar] = y_binary[which.min( rowSums( t(t(Xinput) - Xtest[istar,] )^2 ) )] #euclidean distance
squared,

  }

  y_hat_vec

}

<!-- ``` -->



<!-- 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 yhat randomly. Set the
default `k` to be the square root of the size of 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}

#TO-DO for the 650 students but extra credit for undergrads

```
```