HW02p

Zeke Luger

March 6, 2018

```
knitr::opts_chunk$set(error = TRUE) #this allows errors to be printed into the PDF
```

Welcome to HW02p where the "p" stands for "practice" meaning you will use R to solve practical problems. This homework is due 11:59 PM Tuesday 3/6/18.

You should have RStudio installed to edit this file. You will write code in places marked "TO-DO" to complete the problems. Some of this will be a pure programming assignment. Sometimes you will have to also write English.

The tools for the solutions to these problems can be found in the class practice lectures. I want you to use the methods I taught you, not for you to google and come up with whatever works. You won't learn that way.

To "hand in" the homework, you should compile or publish this file into a PDF that includes output of your code. To do so, use the knit menu in RStudio. You will need LaTeX installed on your computer. See the email announcement I sent out about this. Once it's done, push the PDF file to your github class repository by the deadline. You can choose to make this respository private.

For this homework, you will need the `testthat` libray.

```
if (!require("pacman")){install.packages("pacman")}
```

```
## Loading required package: pacman
```

```
pacman::p_load(testthat, e1071, ggplot2, HistData, modelr)
```

1. Source the simple dataset from lecture 6p:

```
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
)
X_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
y_binary = as.numeric(Xy_simple$response == 1)
```

Try your best to write a general perceptron learning algorithm to the following `Roxygen` spec. For inspiration, see the one I wrote in lecture 6.

```
#' This function implements the "perceptron learning algorithm" of Frank Rosenblatt (1957).
#'
#' @param Xinput      The training data features as an n x (p + 1) matrix where the first column is all
#' @param y_binary    The training data responses as a vector of length n consisting of only 0's and 1'
#' @param MAX_ITER    The maximum number of iterations the perceptron algorithm performs. Defaults to 1
#' @param w           A vector of length p + 1 specifying the parameter (weight) starting point. Defaul
#'                    \code{NULL} which means the function employs random standard uniform values.
#' @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){

    #Algorithm
    for( j in 1:MAX_ITER ) {

        set.seed(19)
```

```
        if (is.null(w)) w = runif(ncol(Xinput))

        for ( i in 1:length(y_binary) ) {

            yhat_i = as.integer( w %*% Xinput[i,] > 0 )

            w = w + as.numeric(y_binary[i] - yhat_i) * Xinput[i,]

        }

    }

    w

}
```

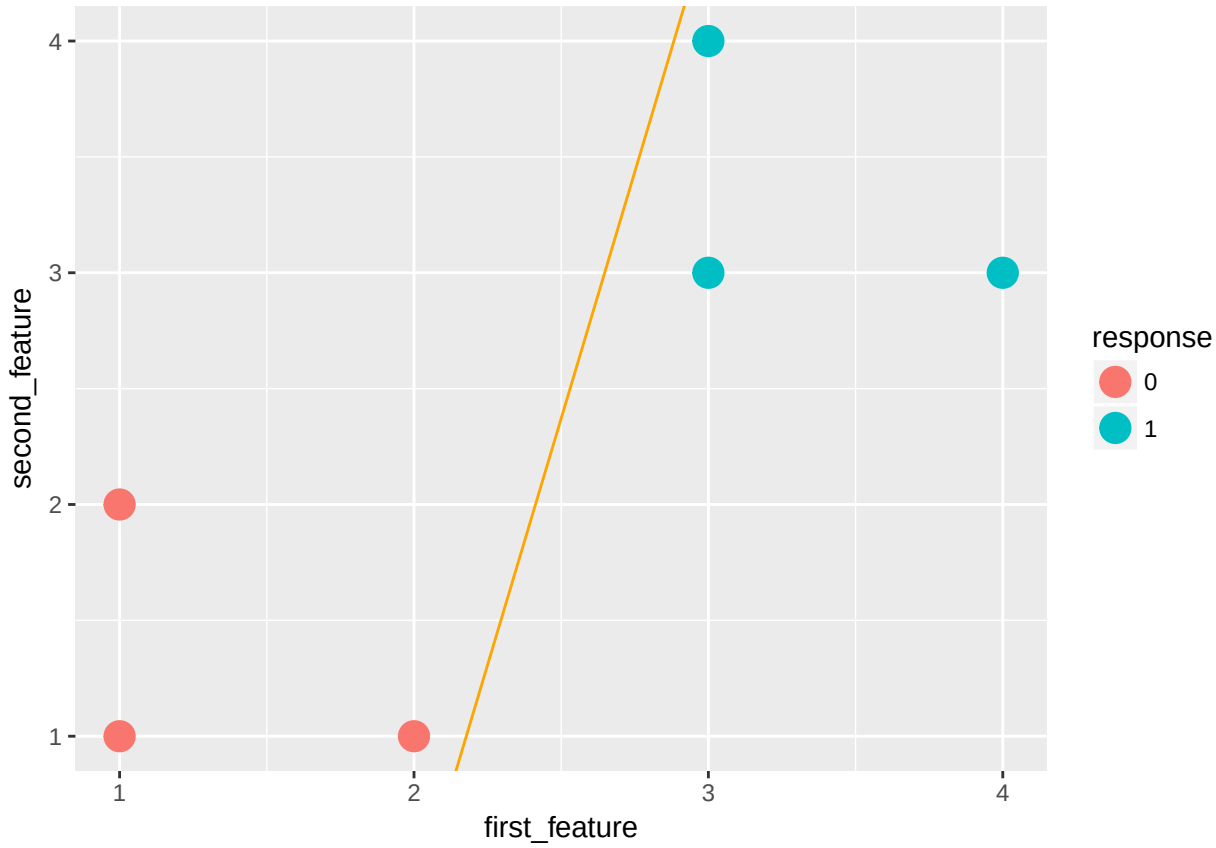
Run the code on the simple dataset above via:

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

```
## [1] -2.882869  1.484030 -0.348793
```

Use the ggplot code to plot the data and the perceptron's g function.

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



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

TO-DO # This line is unsatisfying because it works, but it doesn't seem like the "best" line for classifying data similar to our sample. For example, if there existed another datapoint similar to the one at (3,4), but just to the left of it, this line would mis-classify it.

2. Use the `e1071` package to fit an SVM model to `y_binary` using the predictors found in `X_simple_feature_matrix`. Do not specify the λ (i.e. do not specify the `cost` argument).

```
pacman::p_load(e1071)
p = X_simple_feature_matrix
n = length(y_binary)
lambda = 1e-9
svm_model = svm(Xy_simple, Xy_simple$response, kernel = "linear", cost = (2 * n * lambda)^-1, scale = F
#X_simple_feature_matrix
#Xy_simple
```

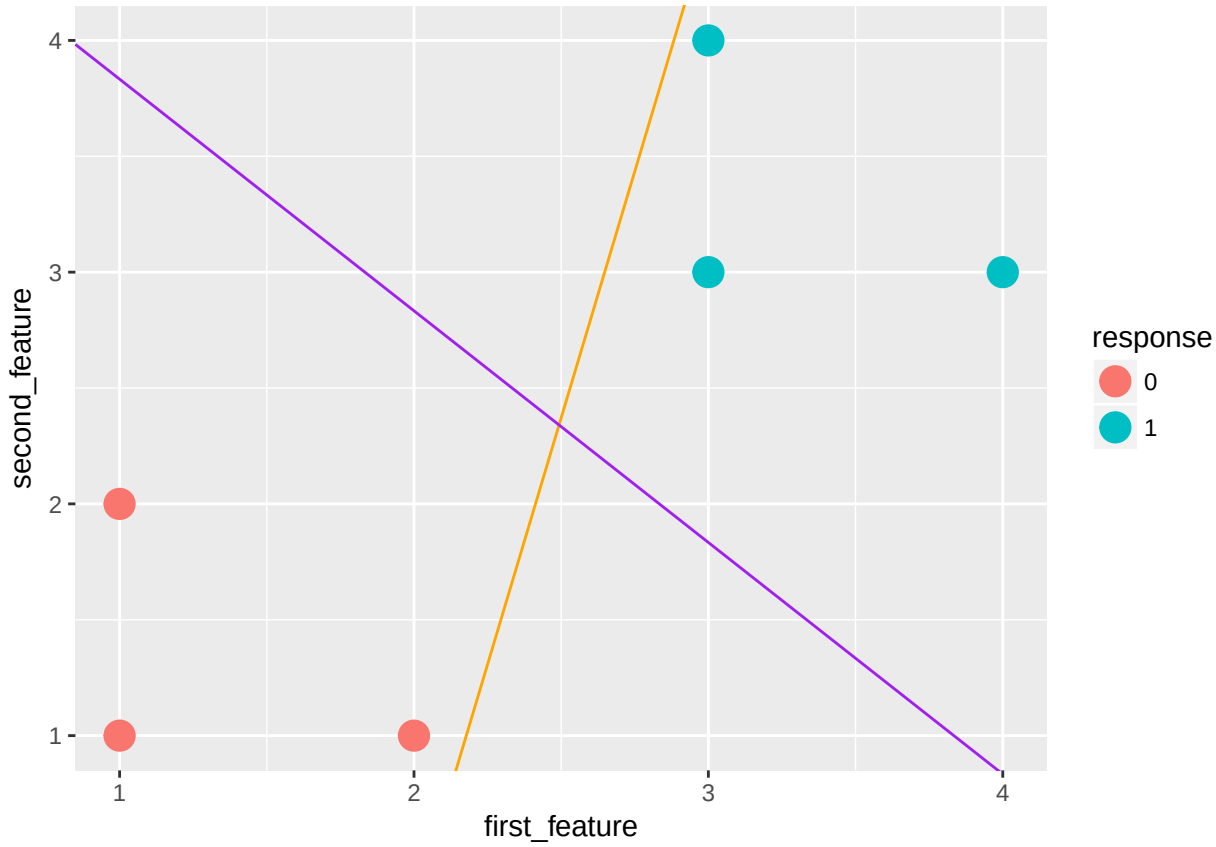
and then use the following code to visualize the line in purple:

```
w_vec_simple_svm = c(
  svm_model$rho, #the b term
  -t(svm_model$coefs) %*% X_simple_feature_matrix[svm_model$index, ] # the other terms
)
w_vec_simple_svm
```

```
## [1] -2.6362526  0.5453318  0.5455247
```

```
simple_svm_line = geom_abline(
    intercept = -w_vec_simple_svm[1] / w_vec_simple_svm[3],
    slope = -w_vec_simple_svm[2] / w_vec_simple_svm[3],
```

```
    color = "purple")
simple_viz_obj + simple_perceptron_line + simple_svm_line
```




```
-w_vec_simple_svm[1] / w_vec_simple_svm[3]
```

```
## [1] 4.832508
```

```
-w_vec_simple_svm[2] / w_vec_simple_svm[3]
```

```
## [1] -0.9996464
```

Is this SVM line a better fit than the perceptron?

TO-DO # It's a much better fit because it evenly divides the area between the 0 and the 1 responses. If we jittered the x inputs, the SVM line would still likely classify them correctly.

3. Now write your own implementation of the linear support vector machine algorithm respecting the following spec making use of the nelder mead `optim` function from lecture 5p. It turns out you do not need to load the package `neldermead` to use this function. You can feel free to define a function within this function if you wish.

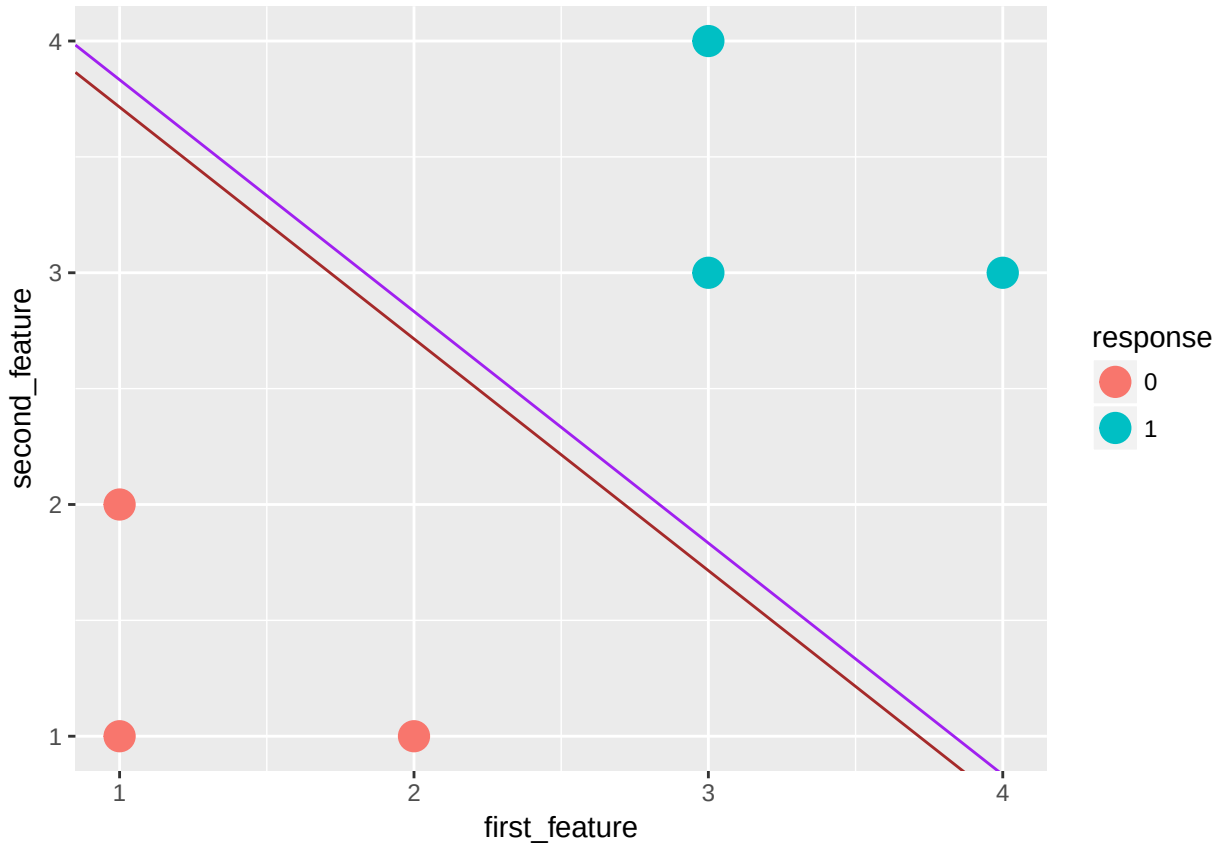
Note there are differences between this spec and the perceptron learning algorithm spec in question #1. You should figure out a way to respect the `MAX_ITER` argument value.

```
#' This function implements the hinge-loss + maximum margin linear support vector machine algorithm of
#'
#' @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'
#' @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
```

```
#'                    The default value is 0.1 to mimic hard margin in the linearly separable case.
#' @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 = 1) {
  #TO-DO

    p = ncol(Xinput)
    n = nrow(Xinput)
    w = numeric( length = p )
    b = numeric( length = 1 )

    weighted_cost = function(par, y, x) {

        b = par[1]
        w = par[2:3]

        zero_vec = rep(0, 6)
        dist = 1/2 - (y - 1/2) * ( w %*% t(x) - b )
        H = mapply(max, zero_vec, dist)

        sum(H) / 2 + lambda * w %*% w

    }

    optim( par = c(0, 0, 0) , fn = weighted_cost, y = y_binary, x = Xinput )

}
```

Run your function using the defaults and plot it in brown vis-a-vis the previous model's line:

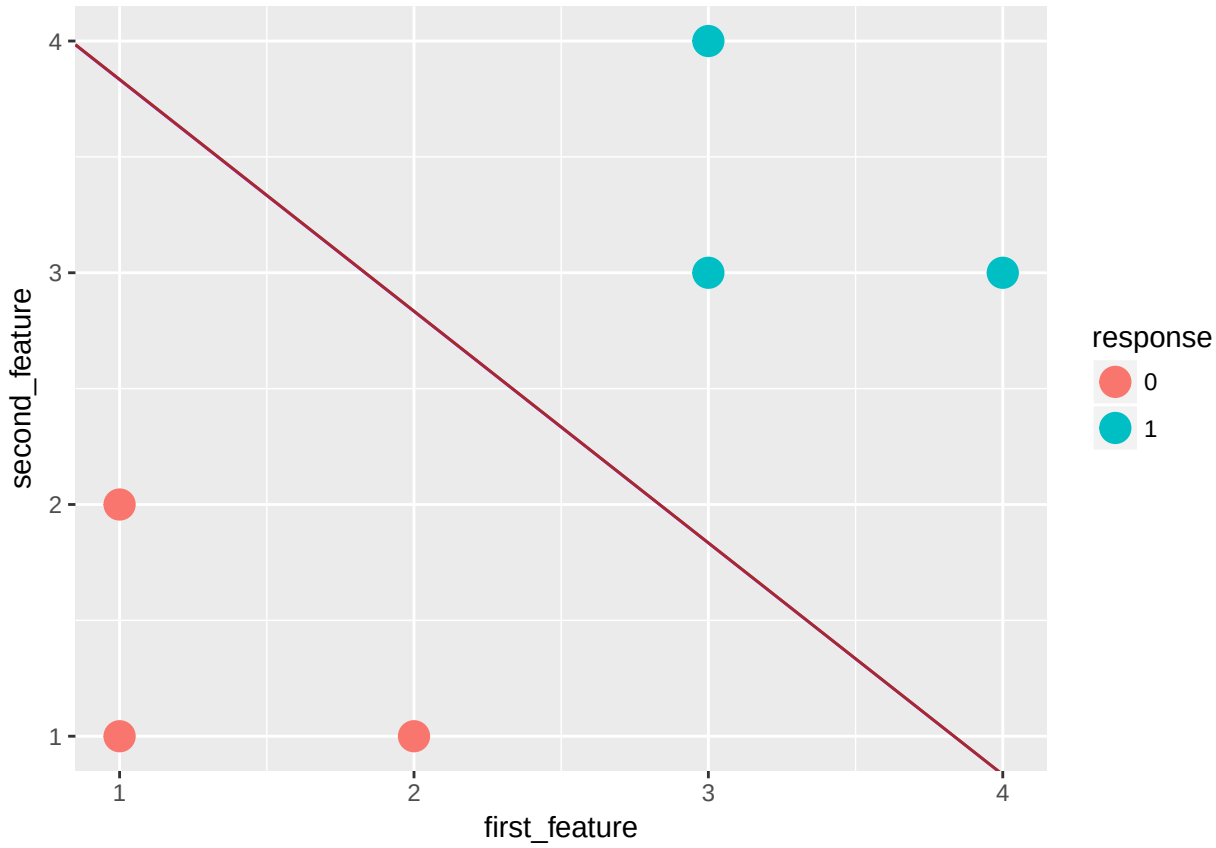
```
svm_model_weights = linear_svm_learning_algorithm(X_simple_feature_matrix, y_binary)
my_svm_line = geom_abline(
    intercept = svm_model_weights$par[1] / svm_model_weights$par[3], #NOTE: negative sign removed from
    slope = -svm_model_weights$par[2] / svm_model_weights$par[3],
    color = "brown")
simple_viz_obj + simple_svm_line + my_svm_line
```



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

It's not exactly the same line. This is probably because e1071 didn't use the same lambda value that we're using. Actually, I tested out some values, and I think e1071 uses .96:

```
svm_model_weights = linear_svm_learning_algorithm(X_simple_feature_matrix, y_binary, lambda = .96)
my_svm_line = geom_abline(
    intercept = svm_model_weights$par[1] / svm_model_weights$par[3], #NOTE: negative sign removed from
    slope = -svm_model_weights$par[2] / svm_model_weights$par[3],
    color = "brown")
simple_viz_obj + simple_svm_line + my_svm_line
```



4. Write a $k = 1$ nearest neighbor algorithm using the Euclidean distance function. Respect the spec below:

```
#' This function implements the nearest neighbor algorithm.
#'
#' @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'
#' @param Xtest        The test data that the algorithm will predict on as a n* x p matrix.
#' @return             The predictions as a n* length vector.
nn_algorithm_predict = function(Xinput, y_binary, Xtest){
  #TO-DO

    numTrain = nrow(Xinput) #n
    numTest = nrow(Xtest) #n*

    minDist = c( rep( Inf, numTest ) )
    argmin = c( rep( NA, numTest))

    # find the nearest neighbor for all Xtest
    # i = the index of the Xtest vector we're testing
    for(i in 1:numTest) {

        # find the nearest neighbor for the each Xtest
        # j = the index of the Xinput (training) vector we're testing
        for(j in 1:numTrain) {

            # minimize distance from Xtest over all the Xinputs
```

```
                distance = sum( ( Xtest[i,] - Xinput[j,] )^2 )

                # return a vector of the indices of the nearest neighbors
                if( distance < minDist[i] ){

                    # update minDist
                    minDist[i] = distance
                    # add the index to the list of y_binary indices to output
                    argmin[i] = j

                }

            }

        }

        # return a vector of y_binarys/yhats of length n*
        y_binary[argmin]

}
```

Write a few tests to ensure it actually works:

```
#TO-DO

data(iris)

# Test if it "predicts" all of the training data
cheating_yHat = nn_algorithm_predict(Xinput = iris[, 1:4 ], y_binary = iris[, 5 ], Xtest = iris[, 1:4 ])
all(cheating_yHat == iris[, 5 ])
```

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

```
# Split the data into train and test data
train =  c( 1:40, 100:140 )
test =  c( 41:50, 141:150 )
yHat = nn_algorithm_predict(Xinput = iris[ train, 1:4 ], y_binary = iris[ train, 5 ], Xtest = iris[ test
all(yHat == iris[ test, 5 ])
```

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

For extra credit, 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.

```
#not required TO-DO --- only for extra credit
#' This function implements the nearest neighbor algorithm.
#'
#' @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'
#' @param Xtest       The test data that the algorithm will predict on as a n* x p matrix.
#' @return            The predictions as a n* length vector.
knn_algorithm_predict = function(k = round(sqrt(length(y_binary))), Xinput, y_binary, Xtest){
  #TO-DO
```

```
    numTrain = nrow(Xinput) #n
    numTest = nrow(Xtest) #n*

    minDist = c( rep( Inf, numTest ) )
    distance = matrix(data = NA, nrow = numTest, ncol = numTrain)
    yHat = y_binary[1]

    # full disclosure: I pulled the function below from the internet
    # because I didn't have time to figure out why
    # R doesn't have a built-in mode function!!!?????
    # https://www.tutorialspoint.com/r/r_mean_median_mode.htm
    modal_value = function(v) {

        values = unique(v)
        values[which.max(tabulate(match(v, values)))]

    }

    # find the nearest neighbor for all Xtest
    # i = the index of the Xtest vector we're testing
    for(i in 1:numTest) {

        # find the k nearest neighbors for the each Xtest
        # j = the index of the Xinput (training) vector we're testing
        # return a matrix of the indices of the nearest neighbors
        for(j in 1:numTrain) distance[ i,j ] = sum( ( Xtest[i,] - Xinput[j,] )^2 )

        # create a vector that is the order of the distance vector for the input vector
        distance_order = order( distance[i, ] )
        # set the argmin = to the first k entries in that vector
        neighbors = y_binary[ distance_order[ 1:k ] ]
        # set yHat vector = mode of neighbors
        yHat[i] = modal_value(neighbors)

    }

    # return a vector of y_binarys/yhats of length n*
    yHat

}
```

A few tests to ensure it actually works:

```
#TO-DO

data(iris)

# Test if it "predicts" all of the training data
cheating_yHat = nn_algorithm_predict(Xinput = iris[, 1:4 ], y_binary = iris[, 5 ], Xtest = iris[, 1:4 ])
all(cheating_yHat == iris[, 5 ])
```

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

```
# Split the data into train and test data
train =  c( 1:40, 100:140 )
test =  c( 41:50, 141:150 )
yHat = nn_algorithm_predict(Xinput = iris[ train, 1:4 ], y_binary = iris[ train, 5 ], Xtest = iris[ tes
all(yHat == iris[ test, 5 ])
```

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

For extra credit, in addition to the argument `k`, add an argument `d` representing any legal distance function to the `nn_algorithm_predict` function. Update the implementation so it performs KNN 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.

```
#not required TO-DO --- only for extra credit
```

5. We move on to simple linear modeling using the ordinary least squares algorithm.

Let's quickly recreate the sample data set from practice lecture 7:

```
n = 20
x = runif(n)
beta_0 = 3
beta_1 = -2
y = beta_0 + beta_1 * x + rnorm(n, mean = 0, sd = 0.33)
```

Solve for the least squares line by computing b_0 and b_1 *without* using the functions `cor`, `cov`, `var`, `sd` but instead computing it from the x and y quantities manually. See the class notes.

```
#TO-DO
b_1 = as.numeric(( x %*% y - n * mean(y) * mean(x) ) / (x %*% x - n * (mean(x)^2) ))
b_0 = as.numeric(mean(y) - b_1 * mean(x))
list(
     "b_0 = " = b_0,
     "b_1 = " = b_1
)
```

```
## $`b_0 = `
## [1] 2.902142
##
## $`b_1 = `
## [1] -1.662427
```

Verify your computations are correct using the `lm` function in R:

```
lm_mod = lm( y ~ x )
b_vec = coef(lm_mod)
expect_equal(b_0, as.numeric(b_vec[1]), tol = 1e-4) #thanks to Rachel for spotting this bug - the b_vec
expect_equal(b_1, as.numeric(b_vec[2]), tol = 1e-4)
```

6. We are now going to repeat one of the first linear model building exercises in history — that of Sir Francis Galton in 1886. First load up package `HistData`.

```
#TO-DO
pacman::p_load(HistData)
```

In it, there is a dataset called `Galton`. Load it using the `data` command:

```
#TO-DO
data(Galton)
```

You now should have a data frame in your workspace called `Galton`. Summarize this data frame and write a few sentences about what you see. Make sure you report n, p and a bit about what the columns represent and how the data was measured. See the help file `?Galton`.

```
#TO-DO
?Galton
list("n = " = nrow(Galton), "p = " = ncol(Galton))
```

```
## $`n = `
## [1] 928
##
## $`p = `
## [1] 2
```

This data come from a study of the heights of 928 adults and the heights of their parents. The columns represent the approximate heights of the parents and children respectively. The parents' heights are measured to the nearest .5 (with an occasional .0), and the chidren's heights are measured to the nearest .2 (with an occasional .7). He also artificially inflated the heights of the female children by a factor of 1.08 in order to homogenize the mixture of male and female heights.

TO-DO

Find the average height (include both parents and children in this computation).

```
avg_height = mean( c(Galton$parent, Galton$child) )
```

Note that in Math 241 you learned that the sample average is an estimate of the "mean", the population expected value of height. We will call the average the "mean" going forward since it is probably correct to the nearest tenth of an inch with this amount of data.

Run a linear model attempting to explain the childrens' height using the parents' height. Use `lm` and use the R formula notation. Compute and report b_0, b_1, RMSE and R^2. Use the correct units to report these quantities.

```
#TO-DO
x = Galton$parent
y = Galton$child
model = lm( y ~ x )
b_0 = as.numeric( model$ coefficients[1] )
b_1 = as.numeric ( model$ coefficients[2] )
list( "b_0 = " = as.numeric( model$ coefficients[1] ),
      "b_1 = " = as.numeric ( model$ coefficients[2] ),
      "RMSE = " = rmse(model, Galton),
      "R^2 = " = rsquare(model, Galton)
      )
```

```
## $`b_0 = `
## [1] 23.94153
##
```

```
## $`b_1 = `
## [1] 0.6462906
##
## $`RMSE = `
## [1] 2.236134
##
## $`R^2 = `
## [1] 0.2104629
```

Interpret all four quantities: b_0, b_1, RMSE and R^2.

TO-DO # For this sample, the expected male child's height is b_0 + b_1 * the parents' average height (24 + .6 * x). That number * .926 gives you the expected height for a female child. # A reasonable estimate of a pseudo-confidence interval for our predicted heights would be ± 2 * RMSE (± 4.5 inches) # Our linear model explains 21% (the R^2 value) of the variance of our sample.

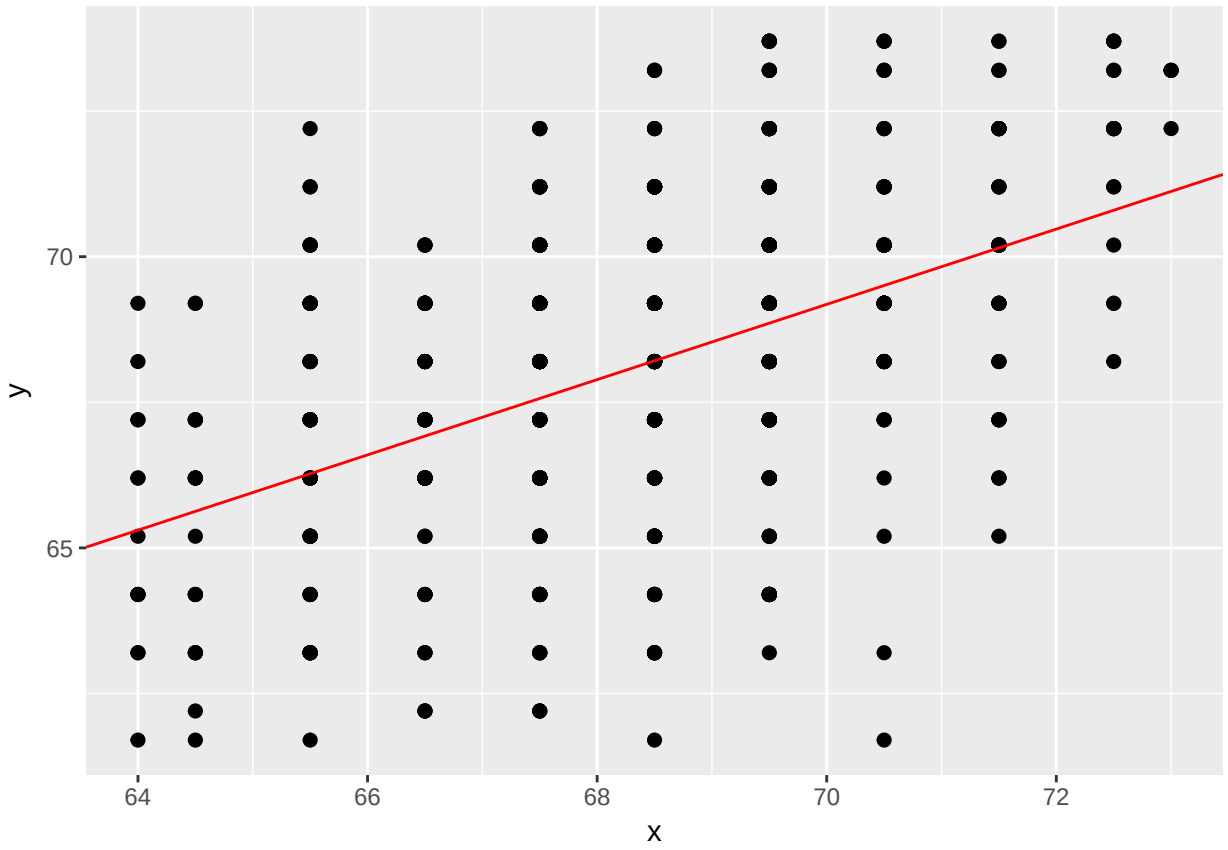
How good is this model? How well does it predict? Discuss.

According to the internet (link below), the standard deviation of single gender human height is 3 inches, which is substantially larger than our RMSE. We've effectively reduced the range of the (pseudo) 95% confidence interval for our sample by ~1/4, from 12 inches to ~9. It reduces the variance by ~20% (R^2), which means it is a poor model, but a significant improvement on the null model. The RMSE pseudo confidence interval is now about 9 inches wide, which isn't all that useful. I don't think prospective parents consulting a genetic counselor would be very impressed by this prediction.

http://www.chegg.com/homework-help/questions-and-answers/problem-3-human-heights-known-normally-distributed-men-mean-height-70-

Now use the code from practice lecture 8 to plot the data and a best fit line using package `ggplot2`. Don't forget to load the library.

```
#TO-DO
pacman::p_load(ggplot2)
simple_df = data.frame(x = x, y = y)
simple_viz_obj = ggplot(simple_df, aes(x, y)) +
  geom_point(size = 2)
simple_ls_regression_line = geom_abline(intercept = b_0, slope = b_1, color = "red")
simple_viz_obj + simple_ls_regression_line
```



It is reasonable to assume that parents and their children have the same height. Explain why this is reasonable using basic biology.

TO-DO # It is reasonable according to the biological theory of evolution. The theory of evolution says that children inherit traits from their parents.

If they were to have the same height and any differences were just random noise with expectation 0, what would the values of β_0 and β_1 be?

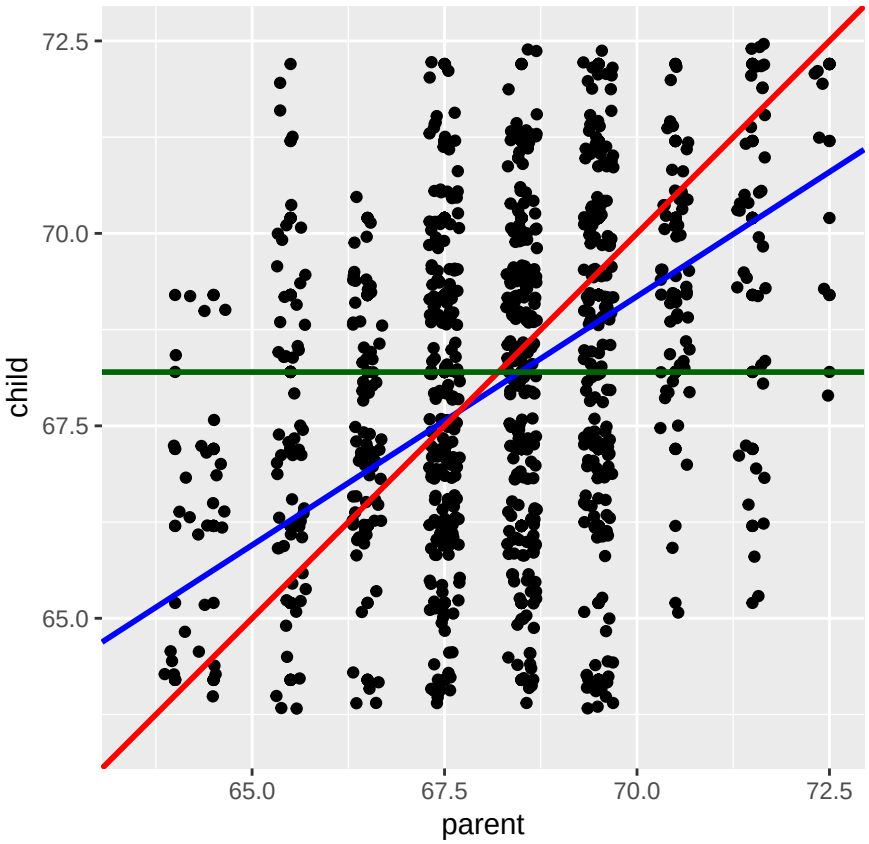
TO-DO # beta_0 would be 0 , and beta_1 would be 1, since the heights would map exactly from parents to child.

Let's plot (a) the data in $\mathbb{D}$ as black dots, (b) your least squares line defined by b_0 and b_1 in blue, (c) the theoretical line β_0 and β_1 if the parent-child height equality held in red and (d) the mean height in green.

```
ggplot(Galton, aes(x = parent, y = child)) +
  geom_point() +
  geom_jitter() +
  geom_abline(intercept = b_0, slope = b_1, color = "blue", size = 1) +
  geom_abline(intercept = 0, slope = 1, color = "red", size = 1) +
  geom_abline(intercept = avg_height, slope = 0, color = "darkgreen", size = 1) +
  xlim(63.5, 72.5) +
  ylim(63.5, 72.5) +
  coord_equal(ratio = 1)
```

```
## Warning: Removed 76 rows containing missing values (geom_point).
```

```
## Warning: Removed 87 rows containing missing values (geom_point).
```



Fill in the following sentence:

TO-DO: Children of short parents became *taller* on average and children of tall parents became *shorter* on average.

Why did Galton call it "Regression towards mediocrity in hereditary stature" which was later shortened to "regression to the mean"?

TO-DO # He called it regression because parents with outlier heights have children who regress towards the mean height for the population.

Why should this effect be real?

TO-DO # This effect could come from the reality that height is a function of an interaction of many genes inherited from both parents. These genes can interact in the child in ways that they did not interact in each parent. However these interactions will regress to the most common interactions among the population. Thus, the children of outlier parents are likely to have more average heights. Also, there could have been environmental factors that impacted the parents' heights, and these factors may not occur for their children. Without that environmental input that moved the parents away from the mean, perhaps the children are more likely to regress to the mean.

You now have unlocked the mystery. Why is it that when modeling with y continuous, everyone calls it "regression"? Write a better, more descriptive and appropriate name for building predictive models with y continuous.

TO-DO # Everyone calls it regression because that's what Galton called it, even though Galton's meaning doesn't usually apply to regression analyses. A more appropriate name might be "linear modeling" since you are building a mathematical model of a relationship between features and responses. Perhaps that's why Professor Kapelner prefers that terminology.