HW02p

Darshan Patel

March 6, 2018

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.

```
knitr::opts_chunk$set(error = TRUE)
```

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

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

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

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

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 1's.
#' @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 perceptron algorithm performs.
#'                    Defaults to 1000.
#' @param w           A vector of length p + 1 specifying the parameter (weight) starting
#'                    point. Default is
#'                    \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){

  if(is.null(w)){
    w = runif(ncol(Xinput))
  }

  for(i in 1:MAX_ITER){
    for(j in 1:nrow(Xinput)){
      x_i = Xinput[j,]
      y_hat_i = ifelse(x_i %*% w > 0, 1,0)
      w = w + as.numeric(y_binary[j] - y_hat_i) * x_i
    }
  }
  w
}
```

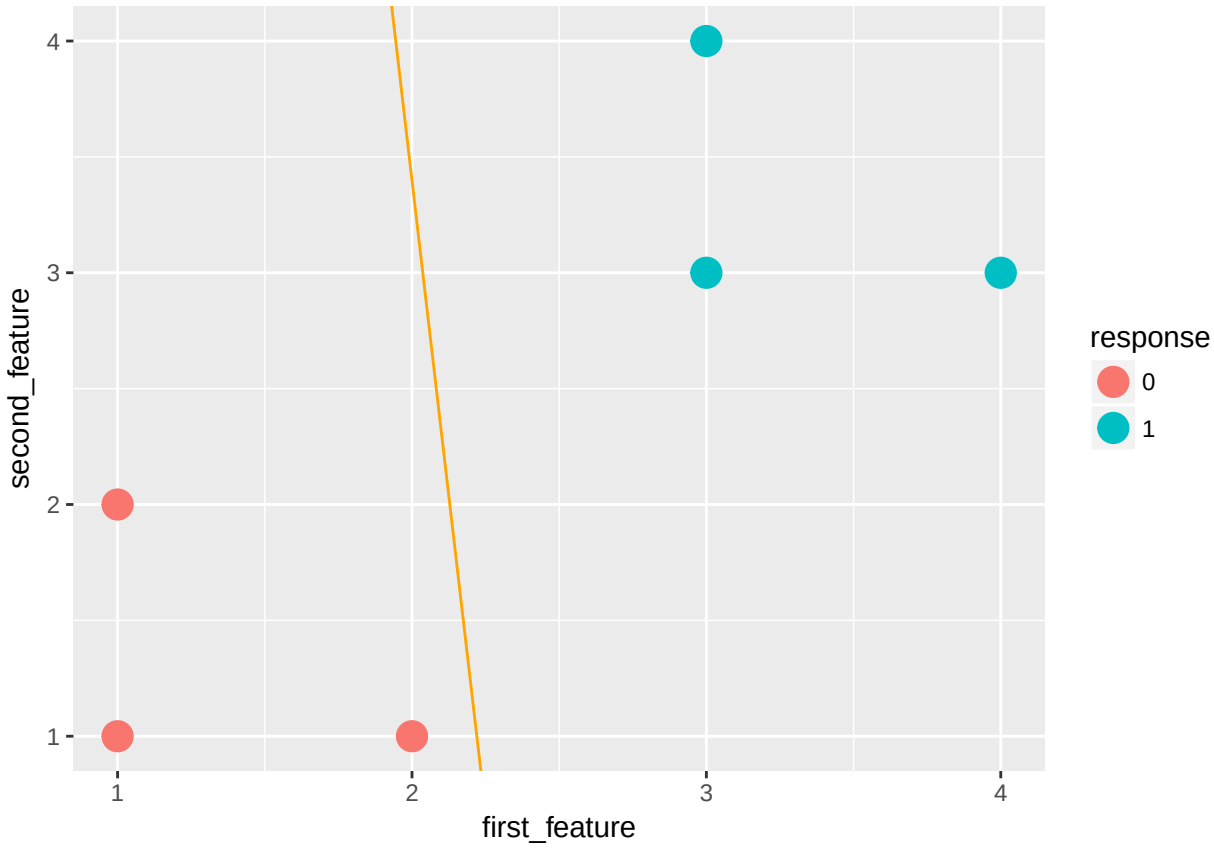
Run the code on the simple dataset above via:

```
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] -10.084341   4.361545   0.400564
```

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?

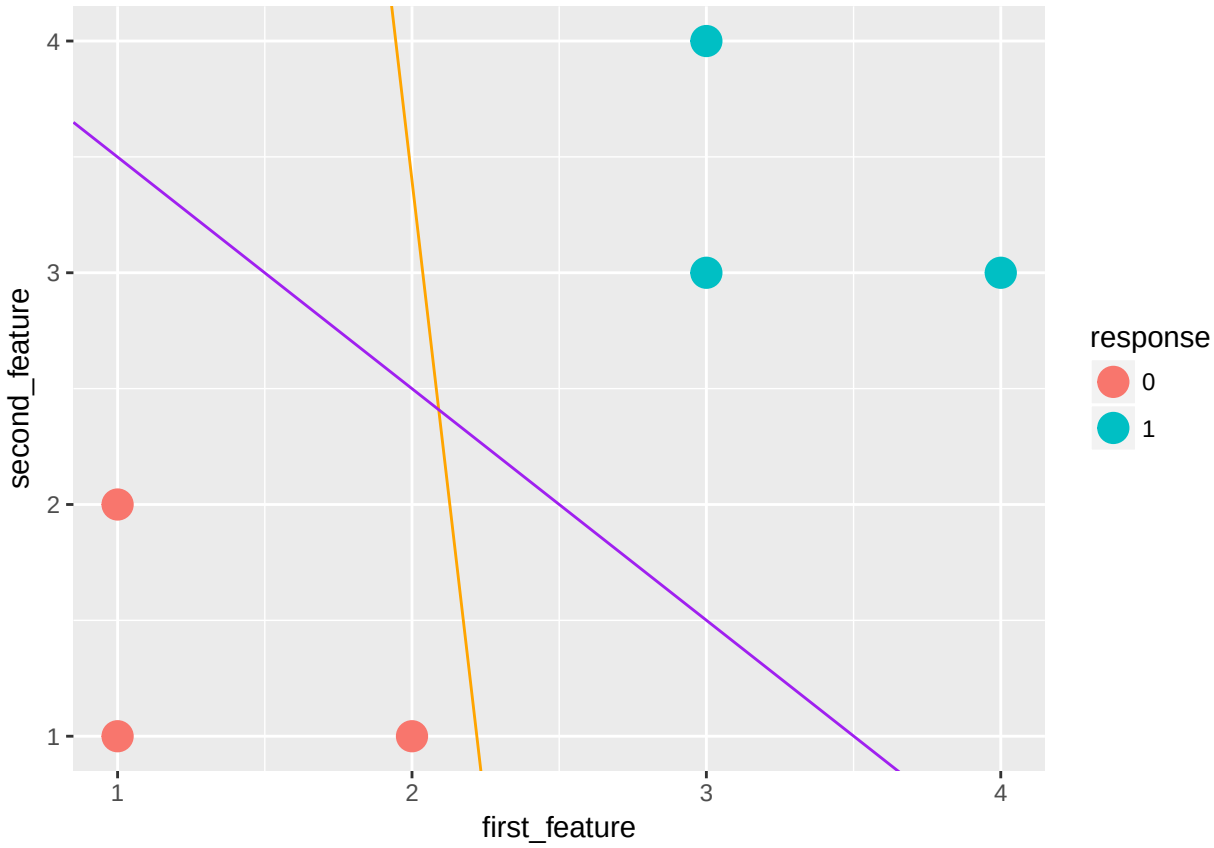
Answer: This line of separation is not "satisfying" because it does not divide the area between the 0s and 1s evenly, or attempt to be somewhat even.

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)
Xy_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
n = nrow(Xy_simple_feature_matrix)
svm_model = svm(Xy_simple_feature_matrix, Xy_simple$response,
                kernel = "linear", scale = FALSE)
```

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
)
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
```



Is this SVM line a better fit than the perceptron?

Answer: The SVM line is a better fit than the perceptron. It seems to divide the area between the 0s and 1s more evenly.

3. Now write pseuocode for 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.

For extra credit, write the actual code.

Answer: Psuedocode Step 1: Add a 1s column to x. Step 2: Initialize w_0 to be a vector of 0s of size $p+1$. Step 3: Run an optimization algorithm on w_0 that computes the minimum avg. hinge loss plus maximum margin according to the constraint where from $i = 1$ to n, the maximum of 2 quantities is computed from all $< x_i, y_i >$ in the data set. Step 4: Repeat step 3 until a maximum iteration is reached.

Extra Credit:

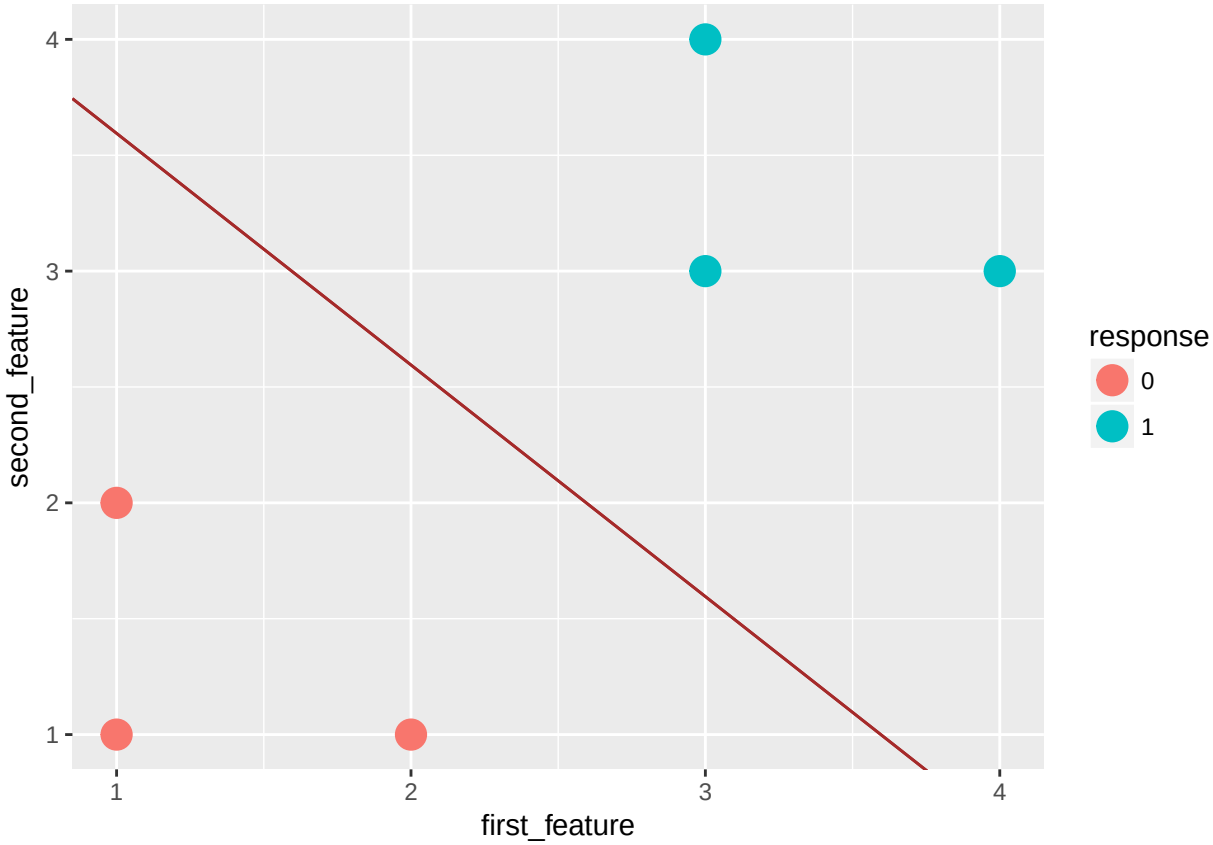
```
#' This function implements the hinge-loss + maximum margin linear support vector machine
#' algorithm of Vladimir Vapnik (1963).
#'
#' @param Xinput      The training data features as an n x p matrix.
#' @param y_binary    The training data responses as a vector of length n consisting of only
#'                    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 0.1.
#' @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){

  p = ncol(Xinput)
  Xinput = cbind(1, Xinput)
  w_0 = rep(0, p+1)
  optim(w_0, function(w){
    hinge_loss = 1 / nrow(Xinput) * sum(pmax(0, 0.5 - (y_binary - 0.5) * (Xinput %*% w)))
    margin_width = lambda * sum(w[2:(p+1)]^2)
    hinge_loss + margin_width
  },control = list(maxit = MAX_ITER))$par
}
```

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:

```
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 + my_svm_line
```



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

Answer: This is the same result from the `e1071` implementation. This makes sense because both algorithms were run on the same concept and optimized the same function.

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's.
#' @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){

  final_ys = rep(NA, nrow(Xtest))
  i_star = NA

  for(i in 1:nrow(Xtest)){
    best_euclid_dist = Inf
    for(j in 1:nrow(Xinput)){
      dist = sqrt(sum((Xinput[i,]- Xtest[i,])^2))
      if(dist < best_euclid_dist){
        best_euclid_dist = dist
        i_star = i
        final_ys[i] = y_binary[i_star]
      }
    }
  }
  final_ys
}
```

Write a few tests to ensure it actually works:

Answer: Use the sample made up dataset. If we test the algorithm on the training data itself, we should expect it to pick the exact points.

```
test = nn_algorithm_predict(X_simple_feature_matrix, y_binary, X_simple_feature_matrix);
expect_equal(test, y_binary)
```

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.

A mode function is first created since R does not provide us with one and we want to use it in the subsequent 2 functions. Source: https://www.r-bloggers.com/computing-the-mode-in-r/

```
#' This function finds the mode of a vector.
#'
#' @param x           The vector of values.
#' @return            The mode of the vector (can be NULL, unimodal, bimodal or multimodal)
mode = function(x){
  ta = table(x)
  tam = max(ta)
  if (all(ta == tam))
    mod = NA
  else{
```

```
    if(is.numeric(x)){
      mod = as.numeric(names(ta)[ta == tam])
    }
    else{
      mod = names(ta)[ta == tam]
    }
  }
  mod
}
```

Implementation of k into $1NN$ algorithm:

```
#' This function implements the nearest neighbor algorithm as well as allow a user to define
#' the number of neighbors required, k.
#'
#' @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 Xtest       The test data that the algorithm will predict on as a n* x p matrix.
#' @param k           The number of neighbors required.
#'                    Default will be square root of size of Xinput
#' @return            The predictions as a n* length vector.
knn_algorithm_predict = function(Xinput, y_binary, Xtest, k = NULL){

  # Computes a default k if needed
  if(is.null(k)){
    k = round(sqrt(nrow(Xinput) * ncol(Xinput)))
  }

  # Predictions
  final_ys = rep(NULL, nrow(Xtest))

  # Stores the y values of the k neighbors
  best_ys = rep(NULL, k)

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

    distance_list = c()

    # Get all distances with respect to 1 point i and store in a vector
    for(j in 1:nrow(Xinput)){
      distance_list[j] = sqrt(sum((Xinput[i,]- Xtest[i,])^2))
    }

    # Pop out the k minimum ones and store the y_binary values
    # at those point into a temp vector
    for(l in 1:k){
      # The which command returns the index of the minimum value in the vector
      # A 1 is placed here in the off chance we get a same min distance at 2 points
      top = which(distance_list == min(distance_list))[1]
      best_ys[l] = y_binary[top]
      distance_list[top] = NULL # Change to NULL so we can get the next min value
    }
```

```
    # The best y value for the x_i point will be the mode of the y_binary values collected
    # If there's no mode or 2+, choose randomly from the points selected or modes respectively
    mode_of_ys = mode(best_ys)
    if(is.na(mode_of_ys)){
      final_ys[i] = sample(best_ys,1)
    }
    if(length(mode_of_ys) > 1){
      final_ys[i] = sample(mode_of_ys,1)
    }
    else{
      final_ys[i] = mode_of_ys
    }
  }
  final_ys
}
```

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.

Note: `d` will be implemented such that it is a function of two parameters that `supposively` comes from somewhere outside this function..

Implementation of `d` into KNN algorithm:

```
#' This function implements the nearest neighbor algorithm as well as allow a user to define
#' the number of neighbors required, k, and the distance formula to use.
#'
#' @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 Xtest       The test data that the algorithm will predict on as a n* x p matrix.
#' @param k           The number of neighbors required.
#'                    Default will be square root of size of Xinput
#' @param d           The formula used to calculate distance between 2 points.
#'                    Default will be the Euclidean distance.
#' @return            The predictions as a n* length vector.
knn_dist_algorithm_predict = function(Xinput, y_binary, Xtest, k = NULL, d = NULL){

  # Computes a default k if needed
  if(is.null(k)){
    k = round(sqrt(nrow(Xinput) * ncol(Xinput)))
  }

  # Configures a default distance formula
  if(is.null(d)){
    d = function(x,y) {
      sqrt(sum((x-y)^2))
    }
  }

  # Predictions
  final_ys = rep(NULL, nrow(Xtest))
```

```
  # Stores the y values of the k neighbors
  best_ys = rep(NULL, k)

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

    distance_list = c()

    # Get all distances with respect to 1 point i and store in a vector
    for(j in 1:nrow(Xinput)){
      distance_list[j] = d(Xinput[i,], Xtest[i,])
    }

    # Pop out the k minimum ones and store the y_binary values
    # at those point into a temp vector
    for(l in 1:k){
      # The which command returns the index of the minimum value in the vector
      # A 1 is placed here in the off chance we get a same min distance at 2 points
      top = which(distance_list == min(distance_list))[1]
      best_ys[l] = y_binary[top]
      distance_list[top] = NULL # Change to NULL so we can get the next min value
    }

    # The best y value for the x_i point will be the mode of the y_binary values collected
    # If there's no mode or 2+, choose randomly from the points selected or modes respectively
    mode_of_ys = mode(best_ys)
    if(is.na(mode_of_ys)){
      final_ys[i] = sample(best_ys,1)
    }
    if(length(mode_of_ys) > 1){
      final_ys[i] = sample(mode_of_ys,1)
    }
    else{
      final_ys[i] = mode_of_ys
    }
  }
  final_ys
}
```

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)
y
```

```
##  [1] 1.3435442 1.5279186 1.2711090 2.4489904 2.7700076 1.4937561 1.8641135
##  [8] 2.1853874 1.0871614 1.7786073 2.3857587 1.6586866 0.8218835 2.5149461
## [15] 1.5861052 2.0863974 2.1417357 1.6143072 2.3517262 3.1193861
```

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.

```
xysum = sum(x*y)
xbar = sum(x) / length(x)
ybar = sum(y) / length(y)
xsqsum = sum(x*x)
w1_num = xysum - (n*xbar*ybar)
w1_denom = xsqsum - (n*xbar*xbar)
b_1 = w1_num / w1_denom
b_0 = ybar - (b_1 * xbar)
b_0
```

```
## [1] 3.202219
```

```
b_1
```

```
## [1] -2.234354
```

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)
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`.

```
pacman::p_load(HistData)
```

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

```
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`.

```
summary(Galton)
```

```
##      parent          child
##  Min.   :64.00   Min.   :61.70
##  1st Qu.:67.50   1st Qu.:66.20
##  Median :68.50   Median :68.20
##  Mean   :68.31   Mean   :68.09
##  3rd Qu.:69.50   3rd Qu.:70.20
##  Max.   :73.00   Max.   :73.70
```

Answer: The Galton dataset presents a table of height relationships between $n = 928$ adult children and 205 parents ($p = 1$). The dataset was constructed by relating each children's height to the average of the father and mother's height. From the shortest child to the tallest child, each parent's height was recorded in ascending order. For the parents' heights, the average was 68.31 inches, varying from 64.00 inches to 73.00 inches. As for the children's heights, the average was 68.09 inches, varying from 61.70 inches to 73.70 inches.

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

```
avg_height = sum(Galton$parent + Galton$child) / (length(Galton$parent)
                                                  + length(Galton$child))
avg_height
```

```
## [1] 68.19833
```

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.

The b_0 and b_1 values are respectively,

```
lmch = lm(Galton$child~Galton$parent, data = Galton)
b_vec = coef(lmch)
b_vec
```

```
##   (Intercept) Galton$parent
##    23.9415302     0.6462906
```

The RMSE, in inches, and R^2 values are respectively,

```
summary(lmch)$sigma
```

```
## [1] 2.238547
```

```
summary(lmch)$r.squared
```

```
## [1] 0.2104629
```

The RMSE, in inches, and R^2 can also be computed manually.

```
yhat = b_vec[1] + b_vec[2] * Galton$parent
error = Galton$child - yhat
sse = sum(error^2)
mse = sse / length(Galton$child)
rmse = sqrt(mse)
rmse
```

```
## [1] 2.236134
```

```
s_sq_p = var(Galton$child)
s_sq_e = var(error)
rsq = (s_sq_p - s_sq_e) / s_sq_p
rsq
```

```
## [1] 0.2104629
```

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

Answer: By running a linear model to explain the children's and parents' height, several statistical values were calculated. It was determined that with an intercept/base value of $b_0 = 23.941$ inches for the parent, the children's height increased as much as $b_1 = 0.646$ inches from the parents' height. The RMSE was determined to 2.238 inches, meaning the difference between the model prediction and actual data had a standard deviation of 2.238 inches. The R^2 value was calculated to be 0.210 which says that only 21.0% of the variance was explained using the model.

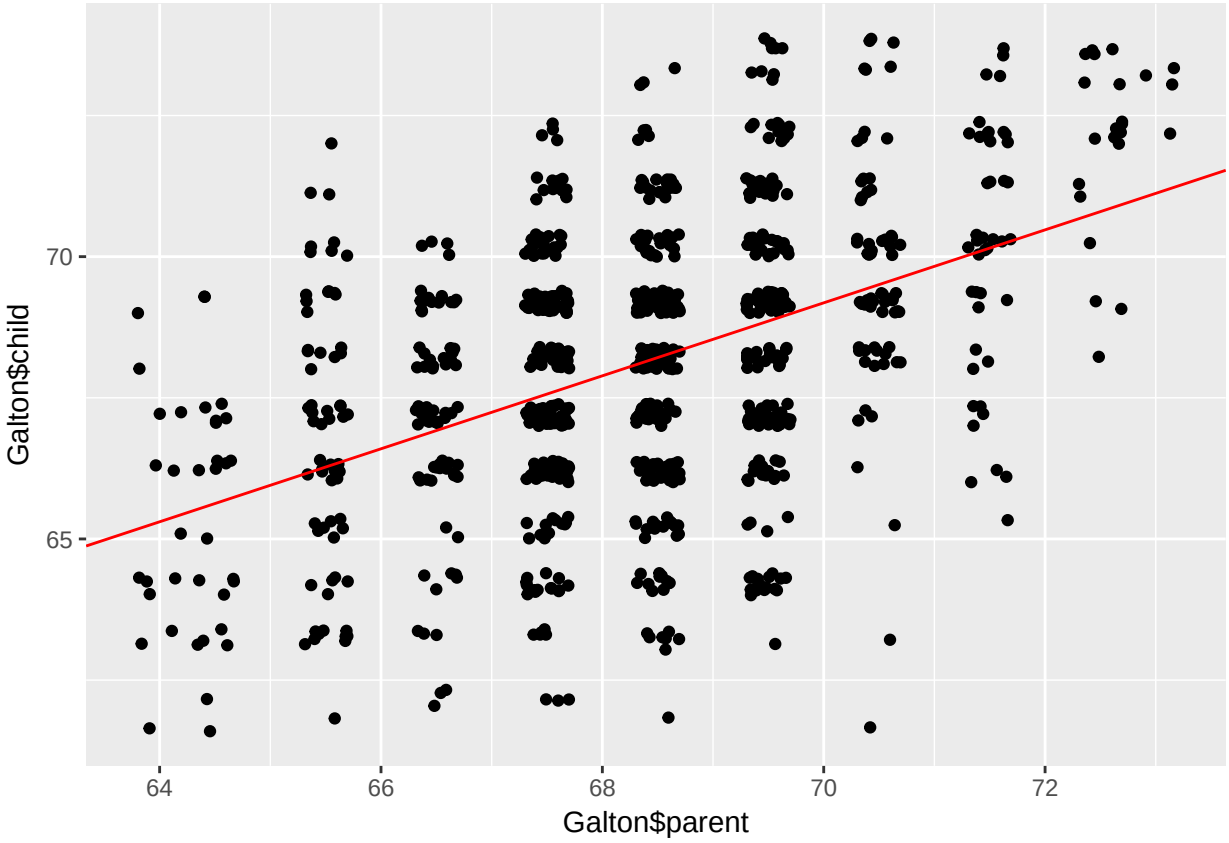
How good is this model? How well does it predict? Discuss.

Seeing that the R^2 value turned out be to be closer to 0 rather than 1, this model is not a good model. Roughly 80% of the variance was unexplained from the regression line. In addition, the RMSE was roughly 2, meaning the difference in height from the regression line and actual data is 2 inches. This makes a hugh difference in height.

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.

Rather than using points, I will use `gitter` to show the denseness of points since we are dealing with $n = 928$ values.

```
pacman::p_load(ggplot2)
data_points = data.frame(x = Galton$parent, y = Galton$child)
data_plot = ggplot(data_points, aes(x = Galton$parent, y = Galton$child)) + geom_jitter()
data_reg = geom_abline(intercept = b_vec[1], slope = b_vec[2], color = "red")
data_plot + data_reg
```



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

Answer: Due to genetics/heredity, it is reasonable to assume that parents and their children will have the same height. A child is formed from DNA, which contains hereditary information from the parents. In the DNA is 23 pairs of chromosomes, each containing some sort of characteristic that can be passed to the child. One of these characteristic is a height attribute. Therefore it is reasonable to assume that if both parents of a child are almost of the same height, the child will also grow to that height at some point.

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?

Answer: The values would be: $\beta_0 = 0$, $\beta_1 = 1$.

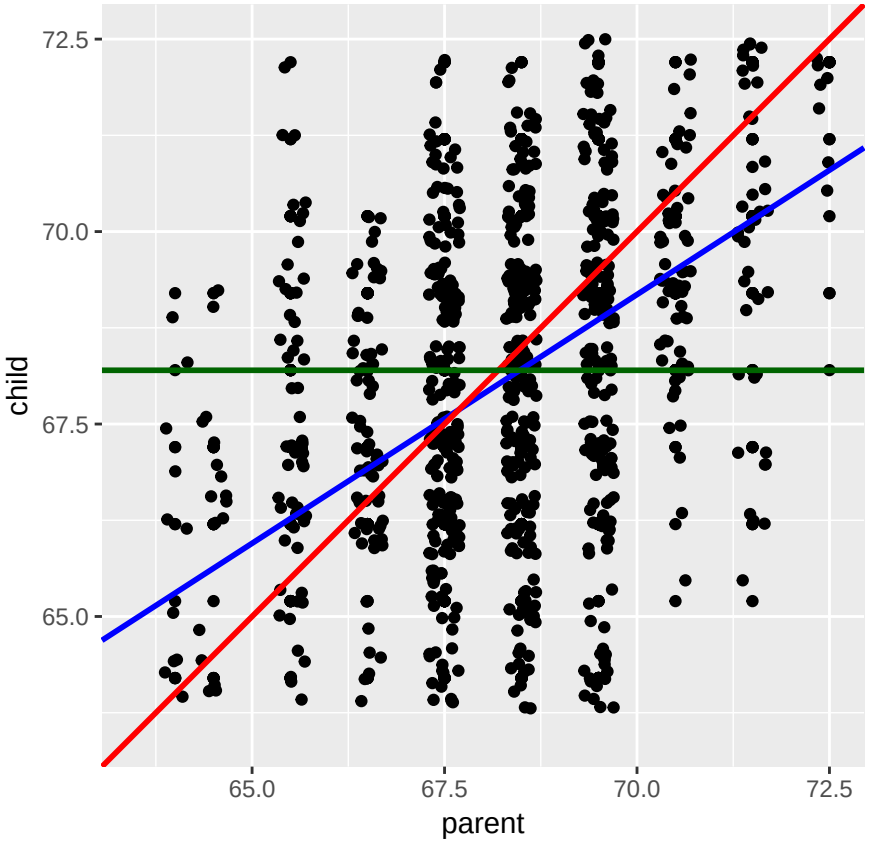
Let's plot (a) the data in $\mathbb{D}$ as black dots, (b) your least squares line defined by b_0 and b_1 in blue, (c) the theoretical line β_0 and β_1 if the parent-child height equality held in red and (d) the mean height in green.

```
ggplot(Galton, aes(x = parent, y = child)) +
  geom_point() +
```

```
  geom_jitter() +
  geom_abline(intercept = b_vec[1], slope = b_vec[2], 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 85 rows containing missing values (geom_point).
```



Fill in the following sentence:

Answer: 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"?

Answer: According to the plot, it is clear that heights did not obey the rules of heredity. Children of tall parents were shorter, and vice versa. In fact, it leaned more to some average height. In other words, it "regressed" to some "mean."

Why should this effect be real?

Answer: This effort should be real because height data of parents comes from 2 different heights. In the data obtained, the average of the parents' height was taken. But in actuality, biology may not even out the playing game and perhaps be biased towards one height than the other (see dominant and regressive genes).

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.

Answer: When modeling with y continuous, it's called regression because we measure the relation between the mean value of the output and the input variables. A better, more descriptive and appropriate name for building predictive models with y continuous would be linear correlation.