HW02p

[Chaim Eisenbach]

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.

```
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
#' @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){
  if (is.null(w)){
    w = runif(ncol(Xinput))
 }
  for (iter in 1 : MAX_ITER){
    for (i in 1 : nrow(Xinput)){
      x_i = Xinput[i, ]
      yhat_i = ifelse(x_i %*% w > 0, 1, 0)
      w = w + as.numeric(y_binary[i] - yhat_i) * x_i
    }
  }
  w
}
  #TO-DO
```

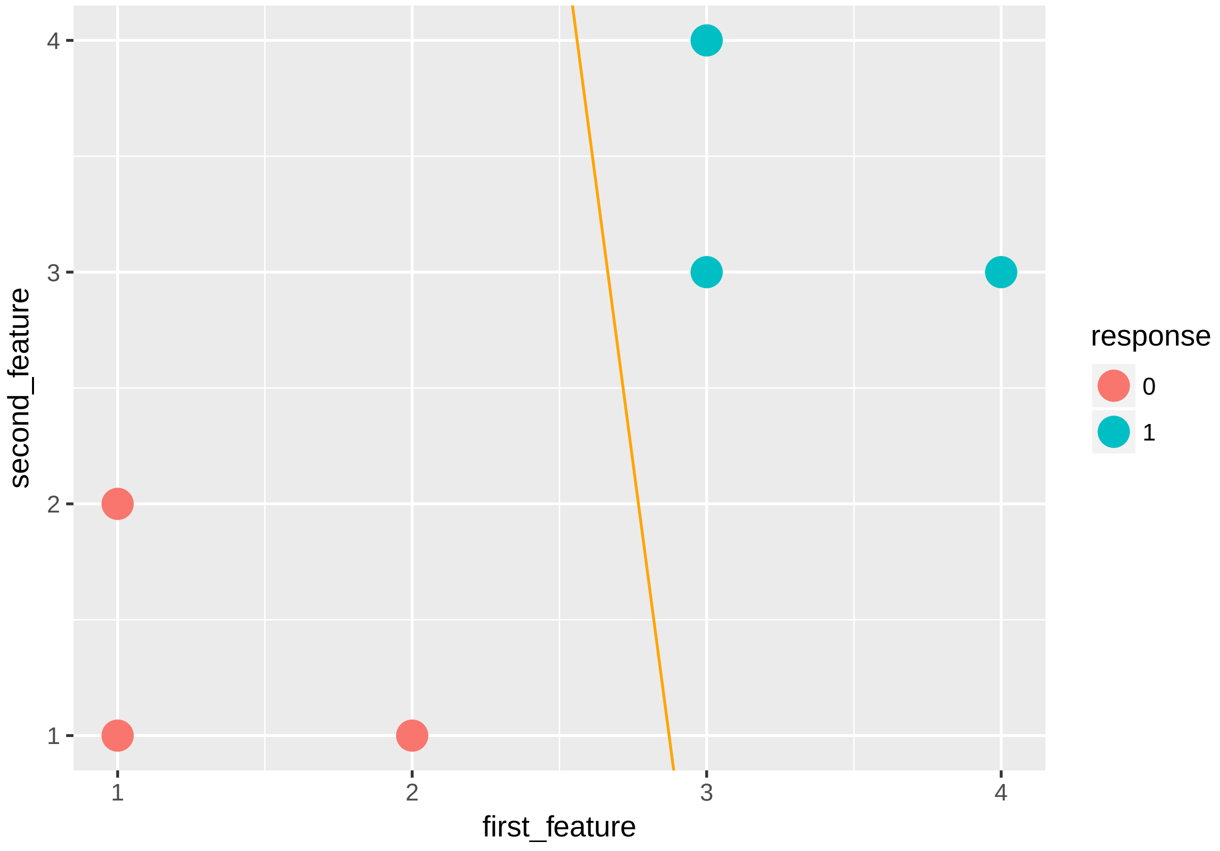
Run the code on the simple dataset above via:

```
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] -9.5868343  3.2207521  0.3353893
```

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?

The line is too close to our values. We would rather have a line "in middle" of both sets of values.

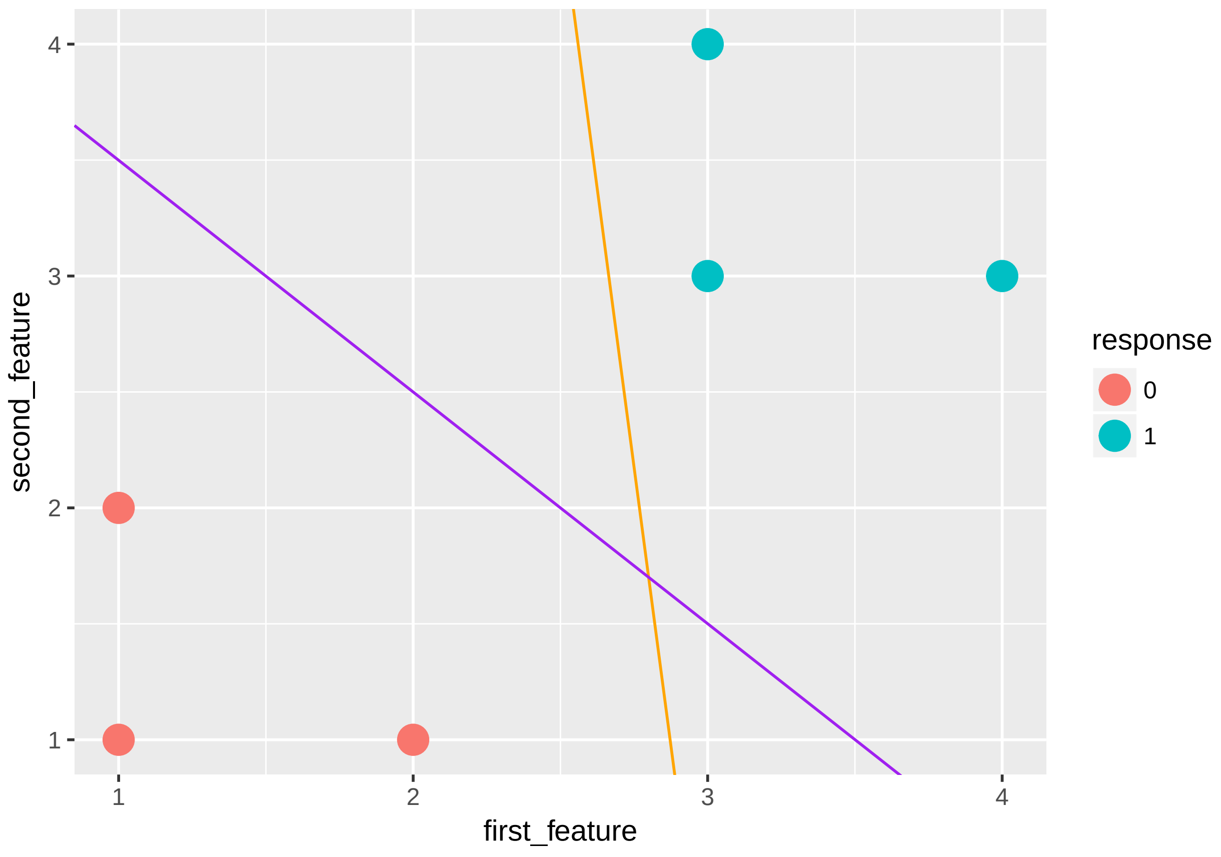
TO-DO

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 lambda

```
pacman::p_load(e1071)
#svm_model = #TO-DO
Xy_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
lambda = 1e-9
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? It looks better. It seperates the data more evenly, as it's seemingly equidistant from the two sets of data. TO-DO

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.

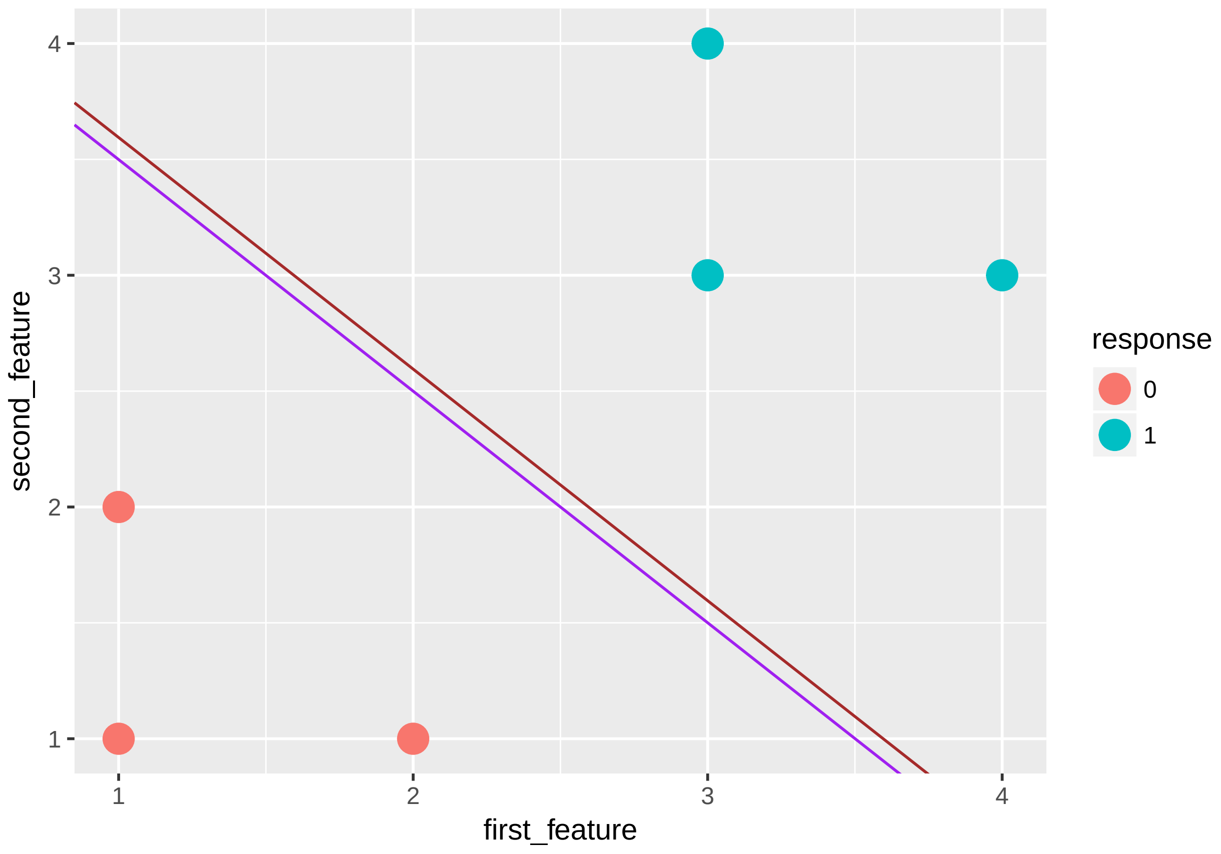
Note there are differences between this spec and the perceptron learning algorithm spec in question #1. You should figure out a way to respect the `MAX_ITER` argument value.

```
#' This function implements the hinge-loss linear support vector machine algorithm of Vladimir Vapnik (
#'
#' @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 .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){
    average_hinge_loss = 1 / nrow(Xinput) * sum(
      pmax(0, 0.5 - (y_binary - 0.5) * (Xinput %*% w)))
    scaled_inverse_margin_width = lambda * sum(w[2 : (p + 1)]^2)
    average_hinge_loss + scaled_inverse_margin_width
  },
  control = list(maxit = MAX_ITER))$par

}
```

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],
    slope = -svm_model_weights[2] / svm_model_weights[3],
    color = "brown")
simple_viz_obj  + my_svm_line + simple_svm_line
```



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

This is not the same. while they have the same slope, however one is closer to one set of data than the other.

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){ #first row, dot with second row
#  #TO-DO
#  best_sqd_distance = 1000
#  i_star = NA
#  for (i in 1 : nrow(Xinput)){
#    eucl = sqrt(((sum((Xinput[i, ] - Xtest[i,])^2))))
#    if (eucl < best_sqd_distance){
#      best_sqd_distance = eucl
#      i_star = best_sqd_distance
#    }
#  }
#  y = i_star
#  y
#}
```

```r
nn_algorithm_predict = function(Xinput, y_binary, Xtest){
  retur=rep(NA,nrow(Xtest))
  best_sqd_distance = 1000
  i_star = NA
    for (i in 1 : nrow(Xtest) ){
      for(index in 1:nrow(Xinput)){
         eucl = sqrt(sum((Xinput[i,] - Xtest[i,])^2))
         if (eucl < best_sqd_distance){
        best_sqd_distance = eucl
        i_star = i
        retur[i]=y_binary[i_star]
         }
      }
    best_sqd_distance= Inf
    }
  retur
}
```

Write a few tests to ensure it actually works:

```r
#TO-DO
nntest = nn_algorithm_predict(X_simple_feature_matrix, y_binary, X_simple_feature_matrix)
expect_equal(nntest,  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.

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

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.

```r
#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:

```r
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.

```r
xbar = mean(x)
ybar = mean(y)
s_x = sqrt(sum((x-mean(x))^2/(length(x)-1)))
s_y = sqrt(sum((y-mean(y))^2/(length(y)-1)))
s_xy = (sum(x*y) - (length(x)*xbar*ybar))/(length(x) -1)
r = (s_xy)/(s_x*s_y)
b_1 = r *(s_y/s_x)
```

```
b_0 = ybar -( b_1 *xbar)
b_1
```

```
## [1] -1.438413
```

```
b_0
```

```
## [1] 2.527304
```

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`.

```
#TO-DO
if (!require("HistData")){install.packages("HistData")}
```

```
## Loading required package: 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`.

```
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
```

This is a set of data that compares the height of a child to their parents. Giving us two columns of height measured in inches. The data seems to say that the childs height is pretty similar to that of the parents, considering that the means are very close. n is 928, and p is 1.

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

```
avg_height = (sum(Galton$parent)+sum(Galton$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.

```r
mod = lm(parent ~ child, data = Galton)
coef(mod)
```

```
## (Intercept)       child
##  46.1353499   0.3256475
```

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

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

```r
summary(mod)$sigma
```

```
## [1] 1.589008
```

```r
b = coef(mod)
```

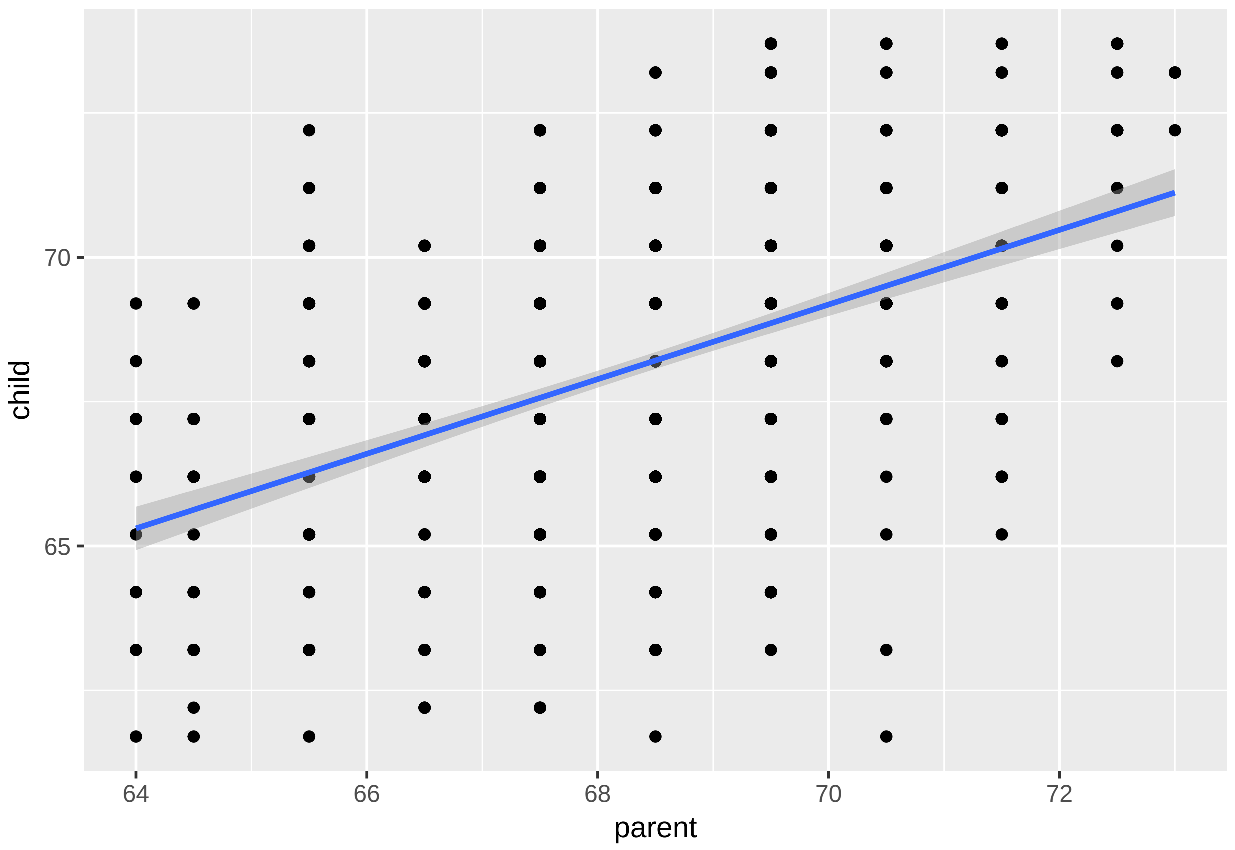
Interpret all four quantities: b_0, b_1, RMSE and R^2. b_0 - The regression seems to say that if the parents height is 0 the childs height would be 46 inches. b_1 - The slope of the model is 0.3256475. R^2 - The parents height only accounts for 21% of the childs height. RMSE - The error between the two data sets is off by around 3 inches in 95% of our predictions.

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

This model doesn't seem very accurate considering the R^2 and RMSE values. 3 inches seems like a pretty big margin considering how close all the heights are in general. And our R^2 also tells us that our model doesn't explain all that much about a childs height based on parents 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.

```r
ggplot(Galton, aes(parent, child)) +
  geom_point() +
  geom_smooth(method = 'lm')
```



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

I think it's reasonable that children would have heights very close to their parents based on genetics alone. However, considering all the other factors in a human beings development including nutrition, it may not be the most comprehensive predictor.

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?

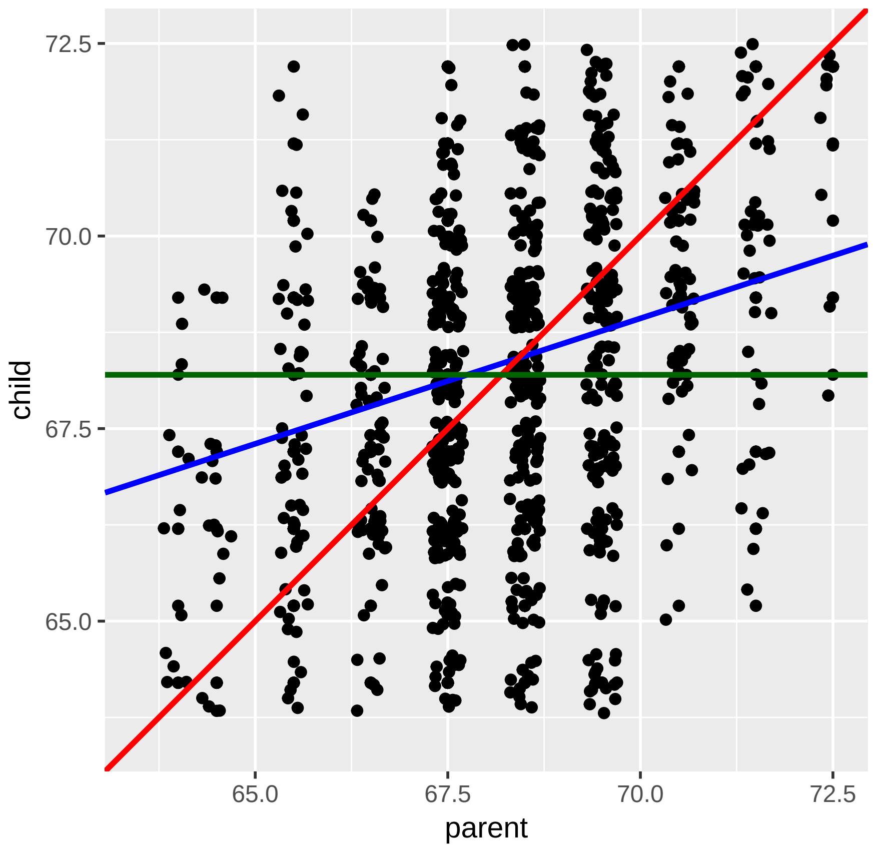
If the heights matched up exactly then x = y, and the intercept would be 0 and the slope would be 1.

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.

```r
ggplot(Galton, aes(x = parent, y = child)) +
  geom_point() +
  geom_jitter() +
  geom_abline(intercept = b[1], slope = b[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:

Children of short parents became short on average and children of tall parents became tall on average.

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

It's unstable in the short run but after a while it will be closer to the average. Even if there are extremely tall or short parents overall, on average, there will be a ''regression to the average 'movement' towards the average.

Why should this effect be real?

Any singular measurement can stand the risk of failing our predictions. Like sometimes guessing the anwers on a test can yield positive or negative results. But done multiple times it will yield 'mediocre' results i.e. you will only get half right if done on a true/false exam.

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.

Estimation or prediction could be better words. They don't make any assumptions about the behavior of the data initially, while Regression seems to imply a very specific thing about the behavior of the data which may or may not be true.