

Lab 3

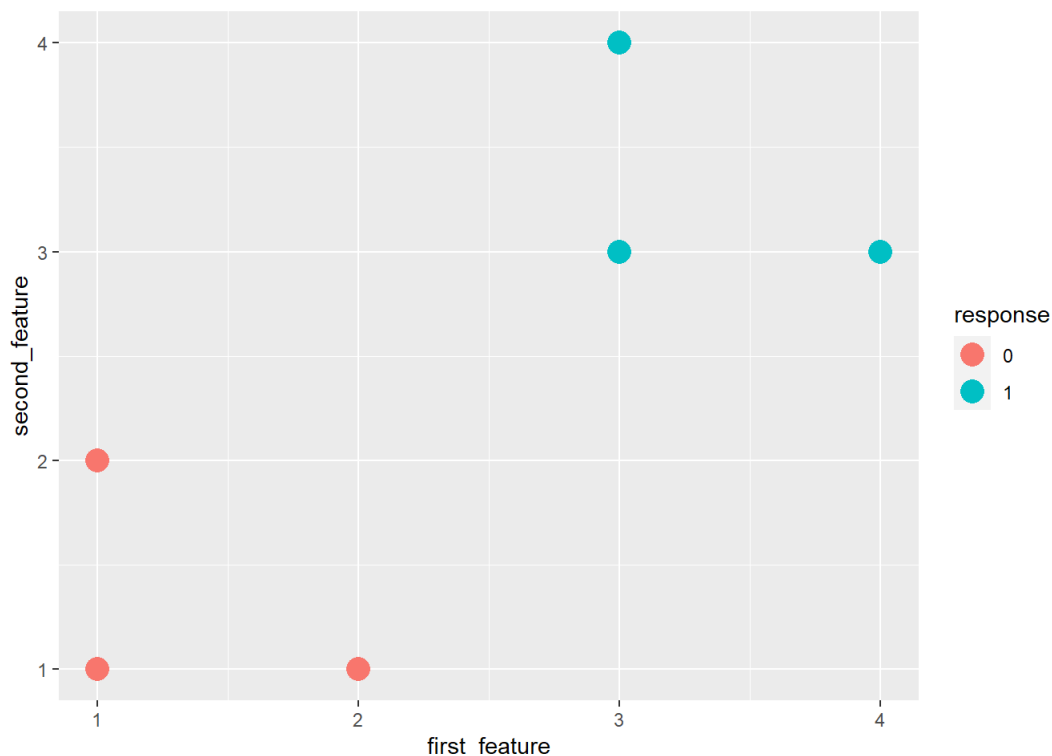
Elizabeth McHugh

11:59PM March 4, 2021

Support Vector Machine vs. Perceptron

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

```
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.

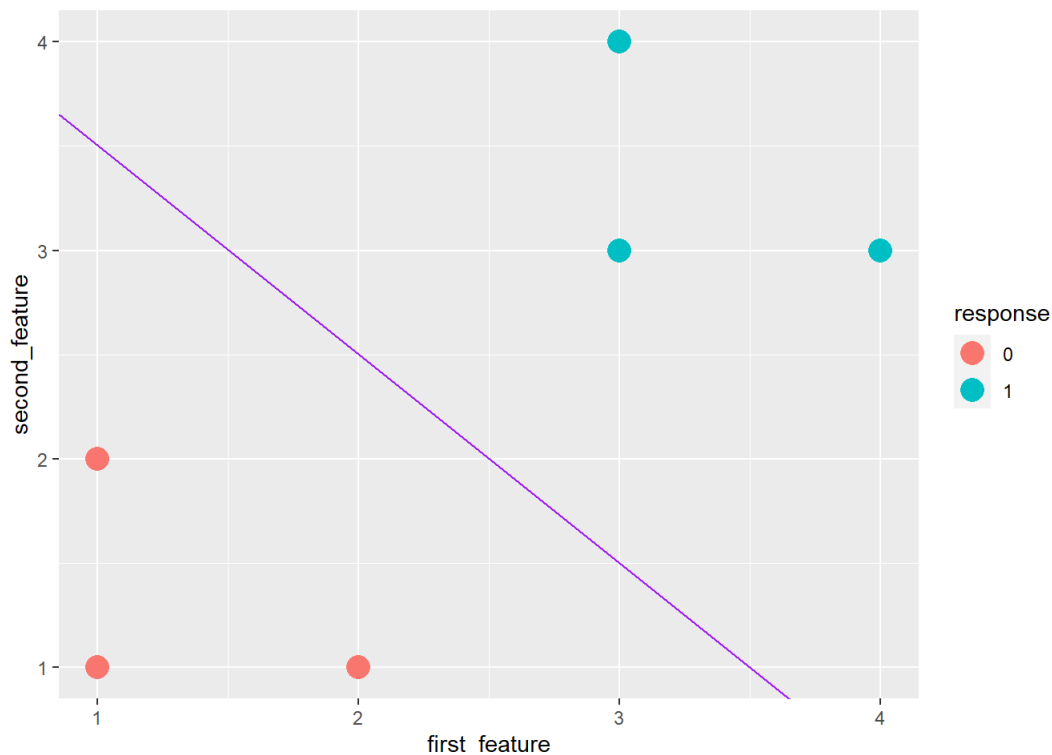
```
pacman::p_load(e1071)

Xy_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
# n = nrow(Xy_simple_feature_matrix)

svm_model = svm(
  Xy_simple_feature_matrix,
  data = 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) %*% cbind(Xy_simple$first_feature, Xy_simple$second_feature)[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_svm_line
```



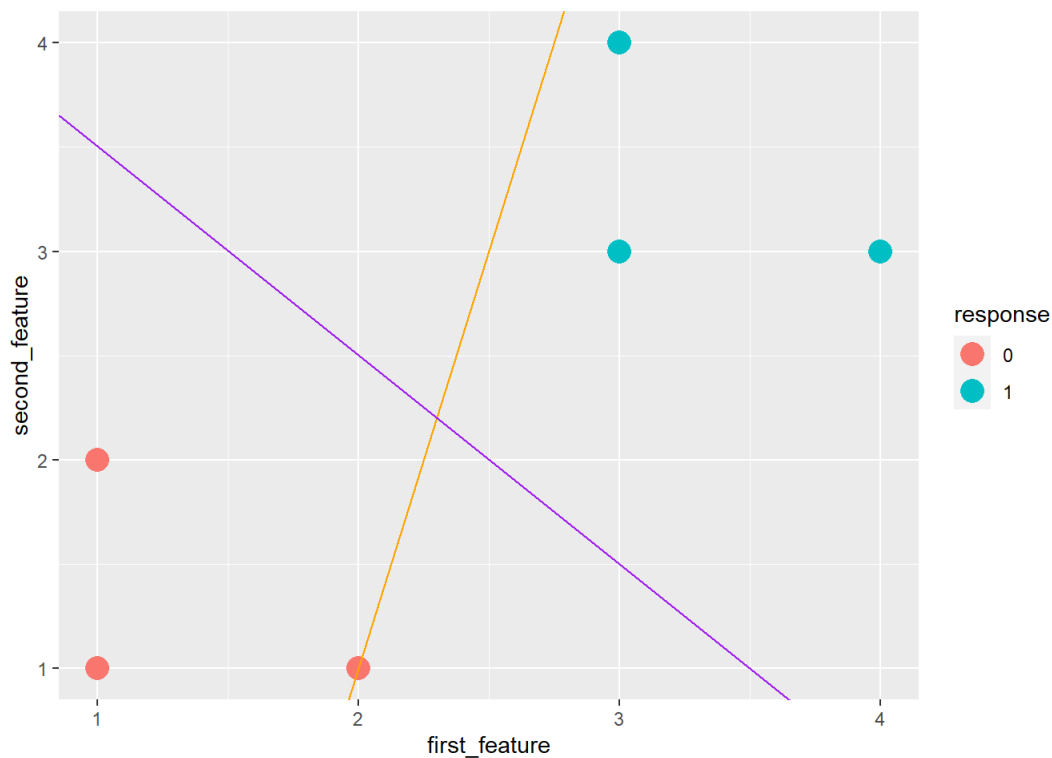
Source the `perceptron_learning_algorithm` function from lab 2. Then run the following to fit the perceptron and plot its line in orange with the SVM's line:

```
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w = rep(0, (ncol(Xinput) + 1))) {
  X = as.matrix(cbind(1, Xinput[, , drop = FALSE]))

  for (i in 1: MAX_ITER) {
    for (j in 1: nrow(X)) {
      x_j = X[j,]
      yhat_j = ifelse(sum(x_j * w) > 0, 1, 0)
      y_j = y_binary[j]

      for (k in 1: ncol(X)) {
        w[k] = w[k] + (y_j - yhat_j) * x_j[k]
      }
    }
  }
  w

  w_vec_simple_per = perceptron_learning_algorithm(
    cbind(Xy_simple$first_feature, Xy_simple$second_feature),
    as.numeric(Xy_simple$response == 1)
  )
  simple_perceptron_line = geom_abline(
    intercept = -w_vec_simple_per[1] / w_vec_simple_per[3],
    slope = -w_vec_simple_per[2] / w_vec_simple_per[3],
    color = "orange")
  simple_viz_obj + simple_perceptron_line + simple_svm_line
```



Is this SVM line a better fit than the perceptron?

Without any more data, this SVM line is a much better fit than the perceptron from lab #2, as it seems to more adequately split the data in half with a more equal buffer around each category for predicting future outcomes.

Now write pseudocode 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.

```
#' 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

  #initialize n and p

  #define SHE

  #for loop to test for min w
  #find w such that (1/n)SHE + lambda * norm_squared(w) is minimized

  #output w
}
```

If you are enrolled in 342W the following is extra credit but if you're enrolled in 650, 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).

```

#' 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 1.0
#' @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:

```

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

We now 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

```

Compute $h^*(x)$ as `h_star_x`, then draw $\epsilon \sim N(0, 0.33^2)$ as `epsilon`, then compute y .

```

h_star_x = beta_0 + beta_1 * x
epsilon = rnorm(n, mean = 0, sd = 0.33)
y = h_star_x + epsilon

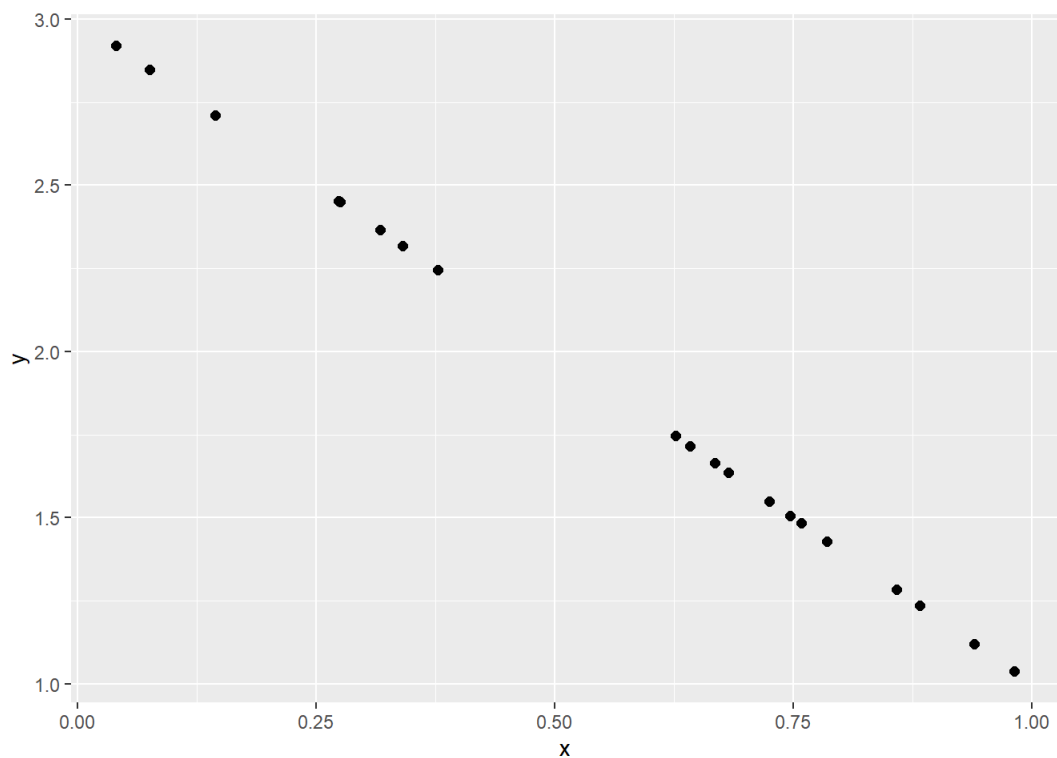
```

Graph the data by running the following chunk:

```

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_viz_obj

```



Does this make sense given the values of β_0 and β_1 ?

This does make sense, based on the given intercept of 3 and slope of -2. #FINISH ME

Write a function `my_simple_ols` that takes in a vector `x` and vector `y` and returns a list that contains the `b_0` (intercept), `b_1` (slope), `yhat` (the predictions), `e` (the residuals), `SSE`, `SST`, `MSE`, `RMSE` and `Rsqr` (for the R-squared metric). Internally, you can only use the functions `sum` and `length` and other basic arithmetic operations. You should throw errors if the inputs are non-numeric or not the same length. You should also name the class of the return value `my_simple_ols_obj` by using the `class` function as a setter. No need to create ROxygen documentation here.

```

my_simple_ols = function(x, y){

  n = length(y)

  if (length(x) != n){
    stop("x and y must be the same length")
  }
  if (class(x) != "numeric" && class(x) != "integer"){
    stop("x must be numeric or integer")
  }
  if (class(y) != "numeric" && class(y) != "integer"){
    stop("y must be numeric or integer")
  }
  if (n <= 2) {
    stop("n must be greater than 2")
  }

  x_bar = sum(x)/n
  y_bar = sum(y)/n

  b_1 = (sum(x*y) - n * x_bar * y_bar) / (sum(x^2) - n * (x_bar)^2)
  b_0 = y_bar - b_1 * x_bar

  y_hat = b_0 + b_1 * x
  e = y - y_hat
  SSE = sum(e^2)
  SST = sum( (y - y_bar)^2)
  MSE = SSE / (n - 2)
  RMSE = sqrt(MSE)
  R_squared = 1 - SSE / SST

  model = list(b_0 = b_0, b_1 = b_1, y_hat = y_hat, e = e, SSE = SSE, SST = SST, MSE = MSE, RMSE = RMSE, R_squared = R_squared)
  class(model) = "my_simple_ols_obj"
  model

}

```

Verify your computations are correct for the vectors `x` and `y` from the first chunk using the `lm` function in R:

```

lm_mod = lm(y ~ x)
my_simple_ols_mod = my_simple_ols(x, y)
#run the tests to ensure the function is up to spec
pacman::p_load(testthat)
expect_equal(my_simple_ols_mod$b_0, as.numeric(coef(lm_mod)[1]), tol = 1e-4)
expect_equal(my_simple_ols_mod$b_1, as.numeric(coef(lm_mod)[2]), tol = 1e-4)
expect_equal(my_simple_ols_mod$RMSE, summary(lm_mod)$sigma, tol = 1e-4)

```

```

## Warning in summary.lm(lm_mod): essentially perfect fit: summary may be
## unreliable

```

```

expect_equal(my_simple_ols_mod$Rsq, summary(lm_mod)$R_squared, tol = 1e-4)

```

```

## Warning in summary.lm(lm_mod): essentially perfect fit: summary may be
## unreliable

```

Verify that the average of the residuals is 0 using the `expect_equal`. Hint: use the syntax above.

```

expect_equal(mean(my_simple_ols_mod$e), 0, tol = 1e-4 )

```

Create the X matrix for this data example. Make sure it has the correct dimension.

```

X = cbind(1,x)
dim(X)

```

```

## [1] 20 2

```

Use the `model.matrix` function to compute the matrix `X` and verify it is the same as your manual construction.

```
model.matrix(~x)
```

```
##      (Intercept)          x
## 1             1 0.64208538
## 2             1 0.88249826
## 3             1 0.66809312
## 4             1 0.07561928
## 5             1 0.37723913
## 6             1 0.98154964
## 7             1 0.85866418
## 8             1 0.72520675
## 9             1 0.93986338
## 10            1 0.14431759
## 11            1 0.27370833
## 12            1 0.03996646
## 13            1 0.78537514
## 14            1 0.27498795
## 15            1 0.74668297
## 16            1 0.62672094
## 17            1 0.34049542
## 18            1 0.68180494
## 19            1 0.75860945
## 20            1 0.31732497
## attr(,"assign")
## [1] 0 1
```

Create a prediction method `g` that takes in a vector `x_star` and `my_simple_ols_obj`, an object of type `my_simple_ols_obj` and predicts `y` values for each entry in `x_star`.

```
g = function(my_simple_ols_obj, x_star){
  my_simple_ols_obj$b_0 + my_simple_ols_obj$b_1 * x_star
}
```

Use this function to verify that when predicting for the average `x`, you get the average `y`.

```
expect_equal(g(my_simple_ols_mod, mean(x)), mean(y))
```

In class we spoke about error due to ignorance, misspecification error and estimation error. Show that as `n` grows, estimation error shrinks. Let us define an error metric that is the difference between b_0 and b_1 and β_0 and β_1 . How about $h = ||b - \beta||^2$ where the quantities are now the vectors of size two. Show as `n` increases, this shrinks.

```
beta_0 = 3
beta_1 = -2
beta = c(beta_0, beta_1) #vector to hold errors
ns = 10^(1:6)
error_in_b = array(NA, length(ns))
for (i in 1 : length(ns)) {
  n = ns[i]
  x = runif(n)
  h_star_x = beta_0 + beta_1 * x
  epsilon = rnorm(n, mean = 0, sd = 0.33)
  y = h_star_x + epsilon

  mod = my_simple_ols(x,y)

  b = c(mod$b_0, mod$b_1)

  error_in_b[i] = sum((beta - b)^2)
}

error_in_b
```

```
## [1] 2.017213e-01 2.891036e-02 9.899778e-04 2.897251e-04 1.558776e-05
## [6] 7.182847e-07
```

```
log(error_in_b, 10)
```

```
## [1] -0.6952482 -1.5389465 -3.0043745 -3.5380138 -4.8072164 -6.1437034
```

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 up.

```
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`. p is 1 and n is 928 the number of observations

```
pacman::p_load(skimr)
skim(Galton)
```

Data summary

Name	Galton
Number of rows	928
Number of columns	2
<hr/>	
Column type frequency:	
numeric	2
<hr/>	
Group variables	None

Variable type: numeric

skim_variable	n_missing	complete_rate	mean	sd	p0	p25	p50	p75	p100	hist
parent	0	1	68.31	1.79	64.0	67.5	68.5	69.5	73.0	
child	0	1	68.09	2.52	61.7	66.2	68.2	70.2	73.7	

TO-DO

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

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

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

If you were predicting child height from parent height and you were using the null model, what would the RMSE of the null model be?

```
sqrt(sum((Galton$child - mean(Galton$child))^2) / (nrow(Galton)-1))
```

```
## [1] 2.517941
```

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 .


```
mod = lm(child ~ parent, Galton)

b_0 = coef(mod)[1]
b_1 = coef(mod)[2]

RMSE = summary(mod)$sigma

r_squared = summary(mod)$r.squared

mod
```

```
##
## Call:
## lm(formula = child ~ parent, data = Galton)
##
## Coefficients:
## (Intercept)      parent
##      23.9415      0.6463
```

```
RMSE
```

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

```
r_squared
```

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

Interpret all four quantities: b_0 , b_1 , RMSE and R^2 . Use the correct units of these metrics in your answer.

b_0 is the “intercept” term, which gives us that if a parent is 0 tall, the child is predicted to be 23.9415 in tall, and the b_1 term gives us that for each centimeter tall the parent is, the child is predicted to be 0.6463 in taller than 23.9415in. For example, for a parent who is 60 in tall, the child will be predicted to be $23.9415 + 60 * 0.6463$ in (or 62.7194 in) tall. The RMSE of mod gives us that the model is off by 2.248547 centimeters on average (within the training data range). The R^2 of 0.2104629 is quite low, which shows us that this might not be the best model for predicting height of children from the height of a parent.

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

The model predicts. (It is, after all, a predictive model.) However, with a low R^2 error, the model might not be expected to be “good”, even though the RMSE of just over 2 centimeters seems be an okay error for prediction. Considering both R^2 and RMSE, I’m not sure I would be too thrilled with using this model to predict the height of my children with much accuracy.

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

Since it is known that genetics influence height, it would be reasonable to assume that children have approximately the same heights of their parents, though each parent likely has a different height.

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?

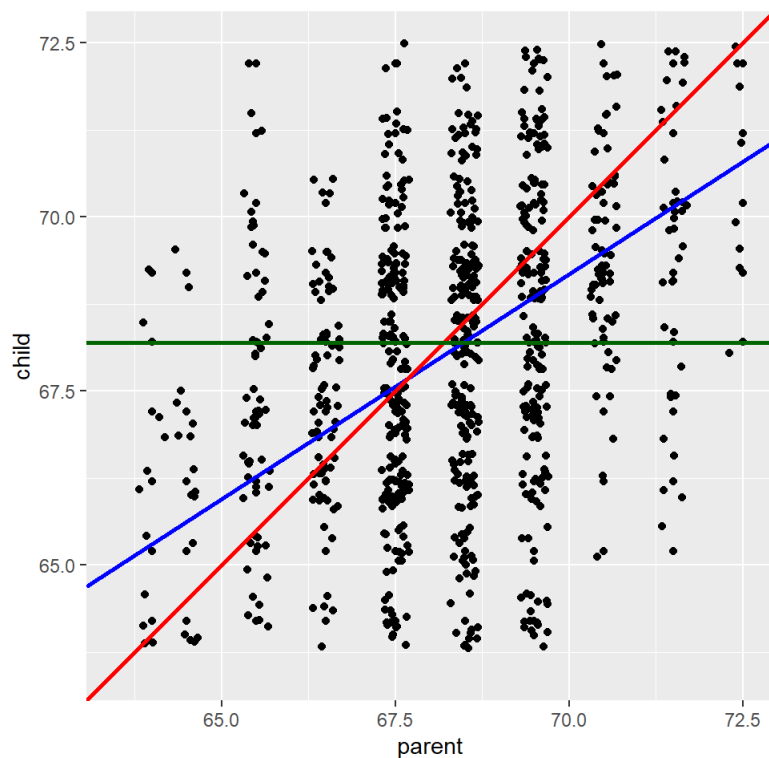
The value of β_0 would be 0, and the value of β_1 would be approximately 1, leaving the predicted height about equal to that of the parent.

Let’s plot (a) the data in 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.

```
pacman::p_load(ggplot2)
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 86 rows containing missing values (geom_point).
```



Fill in the following sentence:

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”?

As time moved on, the exceptionalities within height (shortness or tallness) were “neutralized”, so to speak, bringing both ends of the spectrum closer to the “average” height.

Why should this effect be real?

This effect should be real because nature likes balance.

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.

When modeling with y continuous, the model “draws” predictions towards the mean of the model. A more descriptive and appropriate name for building predictive models with y continuous might be “building a model to predict future values around the linear mean of known data”. However, that would be a horrible name for a modeling process.

You can now clear the workspace. Create a dataset D which we call x_y such that the linear model as R^2 about 50% and RMSE approximately 1.

```
x = c(1:10)
y = c(3, 2, 4, 6, 5, 6, 4, 6, 5, 6)
Xy = data.frame(x = x, y = y)

mod = lm(y ~ x)
summary(mod)$r.squared
```

```
## [1] 0.4702829
```

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

```
## [1] 1.094753
```

Create a dataset D which we call x_y such that the linear model as R^2 about 0% but x , y are clearly associated.

```
x = c(1, 3, 4, 7, 8)
y = x ^ 19
Xy = data.frame(x = x, y = y)

mod = lm(y ~ x)
summary(mod)$r.squared
```

```
## [1] 0.5019083
```

Extra credit: create a dataset D and a model that can give you R^2 arbitrarily close to 1 i.e. approximately $1 - \epsilon$ but RMSE arbitrarily high i.e. approximately M .

```
epsilon = 0.01
M = 1000
#TO-DO
```

Write a function `my_ols` that takes in `X`, a matrix with p columns representing the feature measurements for each of the n units, a vector of n responses `y` and returns a list that contains the `b`, the $p + 1$ -sized column vector of OLS coefficients, `yhat` (the vector of n predictions), `e` (the vector of n residuals), `df` for degrees of freedom of the model, `SSE`, `SST`, `MSE`, `RMSE` and `Rsq` (for the R-squared metric). Internally, you cannot use `lm` or any other package; it must be done manually. You should throw errors if the inputs are non-numeric or not the same length. Or if `X` is not otherwise suitable. You should also name the class of the return value `my_ols` by using the `class` function as a setter. No need to create ROxygen documentation here.

```
my_ols = function(X, y){

  X_1 = as.matrix(cbind(1, X))
  n = length(y)
  p = ncol(X_1)

  if (n != nrow(X)){
    # stop("The length of X and y must be the same.")
  }

  if (class(X[,1]) != "numeric" && class(X) != "integer" && class(X) != "dbl"){
    # stop("The input must be numeric.")
  }

  y_bar = sum(y)/n

  b = solve(t(X_1) %*% X_1) %*% t(X_1) %*% y
  y_hat = X_1 %*% b
  e = y - y_hat
  df = p + 1
  SSE = t(e) %*% e
  SST = t(y - y_bar) %*% (y - y_bar)
  MSE = SSE / (n - df)
  RMSE = sqrt(MSE)
  Rsq = 1 - (SSE / SST)

  model_ols = list("b" = b, "y_hat" = y_hat, "e" = e, "df" = df, "SSE" = SSE, "SST" = SST, "MSE" = MSE, "RMSE" = RMSE, "Rsq" = Rsq)
  class(model_ols) = "my_ols_obj"
  model_ols
}
```

Verify that the OLS coefficients for the `Type` of cars in the cars dataset gives you the same results as we did in class (i.e. the \bar{y} 's within group).

```
X = as.matrix(cars$dist)
y = as.matrix(cars$speed)

my_ols(X, y)
```

```
## $b
##      [,1]
## [1,] 8.2839056
## [2,] 0.1655676
##
## $y_hat
```

```

"""
##          [,1]
## [1,]  8.615041
## [2,]  9.939581
## [3,]  8.946176
## [4,] 11.926392
## [5,] 10.932987
## [6,]  9.939581
## [7,] 11.264122
## [8,] 12.588663
## [9,] 13.913203
## [10,] 11.098554
## [11,] 12.919798
## [12,] 10.601852
## [13,] 11.595257
## [14,] 12.257527
## [15,] 12.919798
## [16,] 12.588663
## [17,] 13.913203
## [18,] 13.913203
## [19,] 15.900014
## [20,] 12.588663
## [21,] 14.244338
## [22,] 18.217960
## [23,] 21.529312
## [24,] 11.595257
## [25,] 12.588663
## [26,] 17.224555
## [27,] 13.582068
## [28,] 14.906609
## [29,] 13.582068
## [30,] 14.906609
## [31,] 16.562284
## [32,] 15.237744
## [33,] 17.555690
## [34,] 20.867041
## [35,] 22.191582
## [36,] 14.244338
## [37,] 15.900014
## [38,] 19.542501
## [39,] 13.582068
## [40,] 16.231149
## [41,] 16.893420
## [42,] 17.555690
## [43,] 18.880230
## [44,] 19.211366
## [45,] 17.224555
## [46,] 19.873636
## [47,] 23.516123
## [48,] 23.681690
## [49,] 28.152015
## [50,] 22.357149
##
## $e
##          [,1]
## [1,] -4.61504079
## [2,] -5.93958139
## [3,] -1.94617594
## [4,] -4.92639228
## [5,] -2.93298684
## [6,] -0.93958139
## [7,] -1.26412199
## [8,] -2.58866258
## [9,] -3.91320318
## [10,] -0.09855441
## [11,] -1.91979773
## [12,]  1.39814831
## [13,]  0.40474287
## [14,] -0.25752743
## [15,] -0.91979773
## [16,]  0.41133742
## [17,] -0.91320318
## [18,] -0.91320318
"""

```

```
## [19,] -2.90001408
## [20,] 1.41133742
## [21,] -0.24433833
## [22,] -4.21796012
## [23,] -7.52931161
## [24,] 3.40474287
## [25,] 2.41133742
## [26,] -2.22455467
## [27,] 2.41793197
## [28,] 1.09339137
## [29,] 3.41793197
## [30,] 2.09339137
## [31,] 0.43771563
## [32,] 2.76225622
## [33,] 0.44431018
## [34,] -2.86704131
## [35,] -4.19158191
## [36,] 4.75566167
## [37,] 3.09998592
## [38,] -0.54250072
## [39,] 6.41793197
## [40,] 3.76885078
## [41,] 3.10658048
## [42,] 2.44431018
## [43,] 1.11976958
## [44,] 2.78863443
## [45,] 5.77544533
## [46,] 4.12636413
## [47,] 0.48387749
## [48,] 0.31830992
## [49,] -4.15201460
## [50,] 2.64285051
##
## $df
## [1,] 3
##
## $SSE
##      [,1]
## [1,] 478.0212
##
## $SST
##      [,1]
## [1,] 1370
##
## $MSE
##      [,1]
## [1,] 10.17066
##
## $RMSE
##      [,1]
## [1,] 3.189148
##
## $Rsq
##      [,1]
## [1,] 0.6510794
##
## attr(,"class")
## [1] "my_ols_obj"
```

```
my_simple_ols(cars$dist, cars$speed)
```

```
## $b_0
## [1] 8.283906
##
## $b_1
## [1] 0.1655676
##
## $y_hat
## [1] 8.615041 9.939581 8.946176 11.926392 10.932987 9.939581 11.264122
## [8] 12.588663 13.913203 11.098554 12.919798 10.601852 11.595257 12.257527
## [15] 12.919798 12.588663 13.913203 13.913203 15.900014 12.588663 14.244338
## [22] 18.217960 21.529312 11.595257 12.588663 17.224555 13.582068 14.906609
## [29] 13.582068 14.906609 16.562284 15.237744 17.555690 20.867041 22.191582
## [36] 14.244338 15.900014 19.542501 13.582068 16.231149 16.893420 17.555690
## [43] 18.880230 19.211366 17.224555 19.873636 23.516123 23.681690 28.152015
## [50] 22.357149
##
## $e
## [1] -4.61504079 -5.93958139 -1.94617594 -4.92639228 -2.93298684 -0.93958139
## [7] -1.26412199 -2.58866258 -3.91320318 -0.09855441 -1.91979773 1.39814831
## [13] 0.40474287 -0.25752743 -0.91979773 0.41133742 -0.91320318 -0.91320318
## [19] -2.90001408 1.41133742 -0.24433833 -4.21796012 -7.52931161 3.40474287
## [25] 2.41133742 -2.22455467 2.41793197 1.09339137 3.41793197 2.09339137
## [31] 0.43771563 2.76225622 0.44431018 -2.86704131 -4.19158191 4.75566167
## [37] 3.09998592 -0.54250072 6.41793197 3.76885078 3.10658048 2.44431018
## [43] 1.11976958 2.78863443 5.77544533 4.12636413 0.48387749 0.31830992
## [49] -4.15201460 2.64285051
##
## $SSE
## [1] 478.0212
##
## $SST
## [1] 1370
##
## $MSE
## [1] 9.958776
##
## $RMSE
## [1] 3.155753
##
## $R_squared
## [1] 0.6510794
##
## attr(,"class")
## [1] "my_simple_ols_obj"
```

```
lm(speed ~ dist, cars)
```

```
##
## Call:
## lm(formula = speed ~ dist, data = cars)
##
## Coefficients:
## (Intercept)      dist
##      8.2839      0.1656
```

Create a prediction method `g` that takes in a vector `x_star` and the dataset `D` i.e. `X` and `y` and returns the OLS predictions. Let `x` be a matrix with `p` columns representing the feature measurements for each of the `n` units

```
#n = 8
#X = matrix(data = NA, nrow = n, ncol = length(y))

g = function(x_star, X, y){
  mod = my_ols(X, y)
  b = mod$b
  x_star = cbind(1, x_star)
  y_hat_star = x_star %*% b
}
```

Processing math: 100%