Lab 3

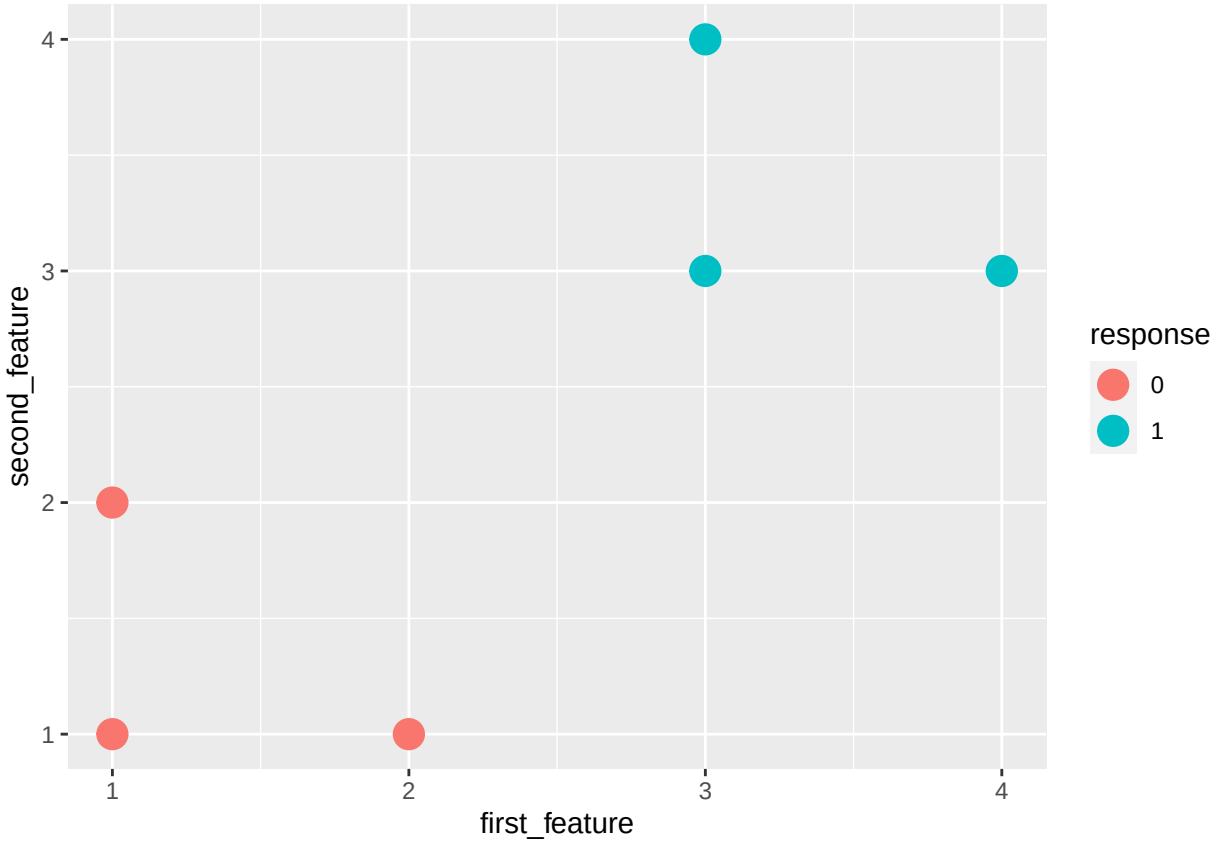
Jia Yu Lin

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.

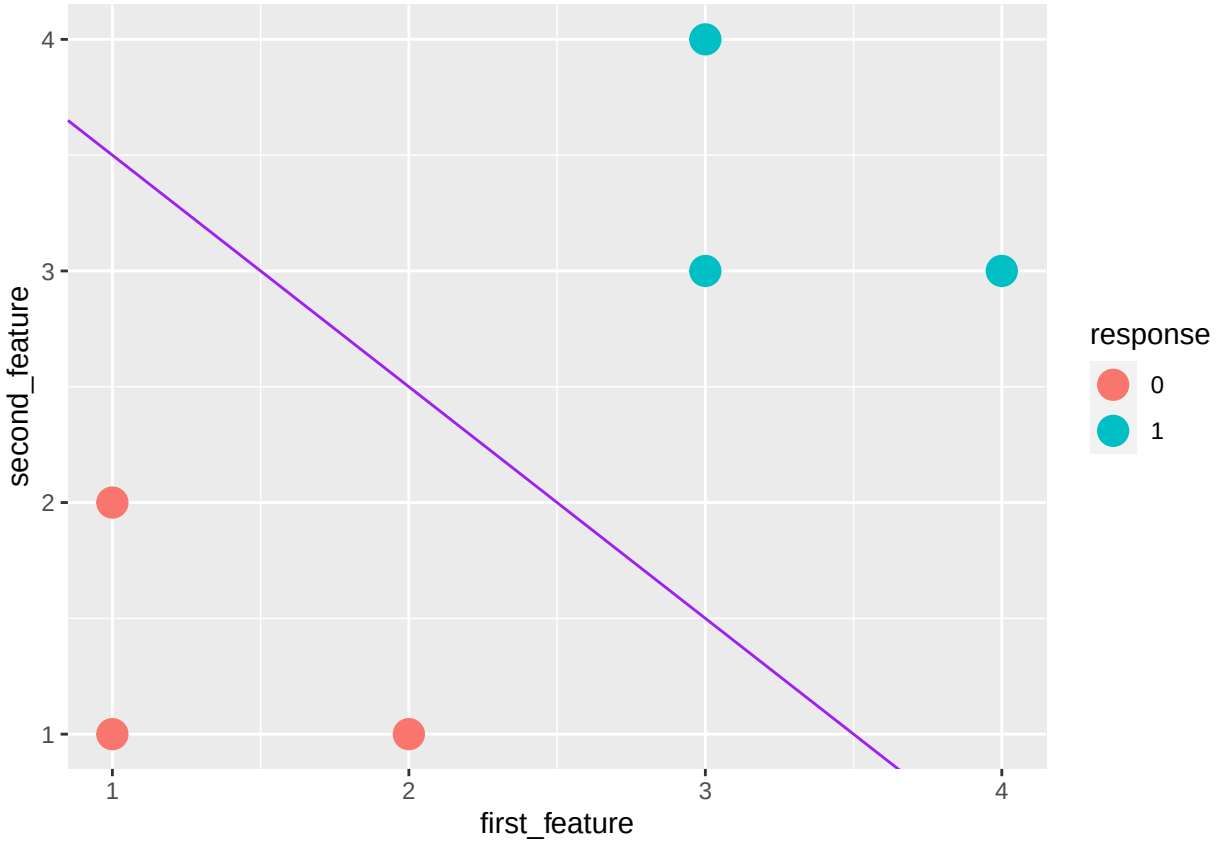
```
pacman::p_load(e1071)
Xy_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
Xy_simple_feature_matrix
```

```
##      first_feature second_feature
## [1,]             1              1
## [2,]             1              2
## [3,]             2              1
## [4,]             3              3
## [5,]             3              4
## [6,]             4              3
```

```
svm_model = svm(
  formula = Xy_simple_feature_matrix,
  x = 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, ] #
)
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:

```
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?

TO-DO Yes because it is separating the points in the data set unlike the perceptron which is going through one of the points.

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.

```
#' Support Vector Machine
#
#' 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 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

  #create a variable called argmin_w_vector
  #loop from 1 to MAX_ITER
    #create variable called min_w_vector
    #create variable called SHE
    #loop from 1 to n
        #find w_Vector and b (y_i - 1/2)(w_vector * x_i -b)>=1/2 using quadratic programming
        #find ||w_vector||
        #compare current magnitude of w_vector to the min_w_vector and set min_w_vector to the new mini
    #loop from 1 to n
      #SHE =  SHE + max{0, (1/2) - (y_i - 1/2)(w_vector * x_i - b)}
    #compare argmin_w_vector with (1/n)SHE + lambda*||min_w_vector||^2 and set argmin_w_vector to the n
  }
```

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
#'
#' @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){
  #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
    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:

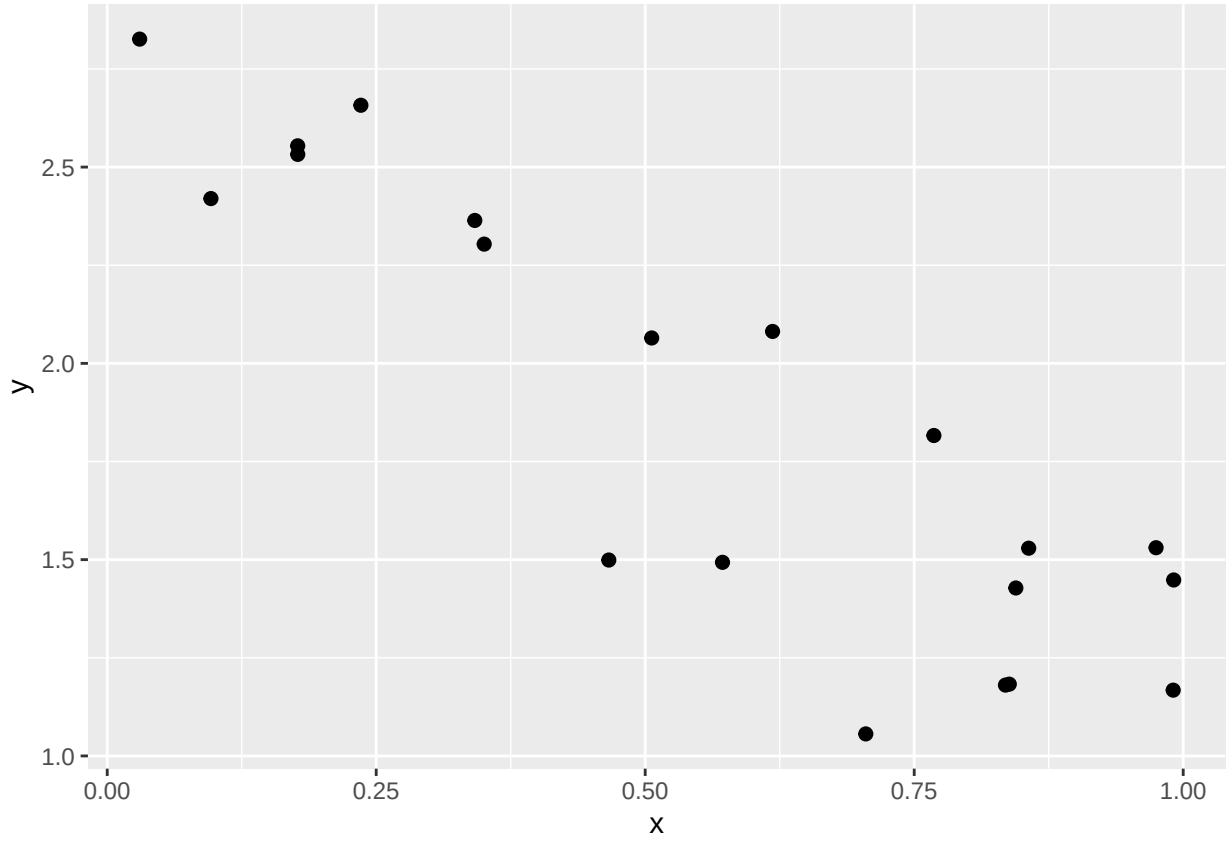
```
n = 20
x = runif(n)
beta_0 = 3
beta_1 = -2
```

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 $beta_0$ and $beta_1$?

TO-DO

Yes, because beta_0 is 3 and beta_0 is the intercept. The slope is around -2 and beta_1 which represents the slope is -2.

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 `Rsq` (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.

```
#TO-DO
my_simple_ols = function(x, y){
  n = length(y)
  if (length(x) != n){
    stop("x and y needs to be same length.")
  }
  if(class(x) != "numeric" && class(x) != "integer"){
    stop("x needs to be numeric.")
  }
  if(class(y) != "numeric" && class(y) != "integer"){
    stop("y needs to be numeric.")
   }
  if( n <= 2){
    stop("n must be more 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
  yhat = b_0 + b_1*x
  e = y - yhat
  SSE = sum(e^2)
  SST = sum((y - y_bar)^2)
  MSE = SSE / (n - 2)
  RMSE = sqrt(MSE)
  Rsq = 1 - (SSE / SST)

  model = list(b_0 = b_0, b_1 = b_1, yhat = yhat, e = e, SSE = SSE, SST = SST, MSE = MSE, RMSE = RMSE, R

  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:

```
#TO-DO
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)
expect_equal(my_simple_ols_mod$Rsq, summary(lm_mod)$r.squared, tol = 1e-4)
```

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

```
#TO-DO
#mean(my_simple_ols_mod$e,0) should not do this because average function can be close to 0, not 0.
mean(my_simple_ols_mod$e)
```

```
## [1] -1.665335e-16
```

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

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

```
#TO-DO
X = cbind(1,x)
```

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

```
#TO-DO
model.matrix(~x)
```

```
##    (Intercept)          x
## 1            1 0.57183454
## 2            1 0.99071193
## 3            1 0.50602021
## 4            1 0.83480500
## 5            1 0.97477813
## 6            1 0.35035568
## 7            1 0.17707149
## 8            1 0.09637763
## 9            1 0.03013028
## 10           1 0.85641330
## 11           1 0.34161402
## 12           1 0.99116097
## 13           1 0.84447515
## 14           1 0.61839095
## 15           1 0.83836713
## 16           1 0.17702600
## 17           1 0.70499624
## 18           1 0.46616951
## 19           1 0.76829023
## 20           1 0.23574144
## 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){
  #TO-DO
  y_star = my_simple_ols_obj$b_0 + my_simple_ols_obj$b_1 * x_star
  y_star

  #if you predicted things correctly, you should get y_bar [video]
}
```

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)

ns = 10*(1:8)#TO-DO
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

  #TO-DO
  mod = my_simple_ols(x,y)
  #order matters here because of line 256
  b = c(mod$b_0, mod$b_1)
  error_in_b[i] = sum((beta - b)^2)
}
error_in_b
```

```
## [1] 0.019311120 0.063585527 0.057122950 0.013773012 0.001416069 0.020922517
## [7] 0.012470304 0.109402028
```

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

```
## [1] -1.7141925 -1.1966417 -1.2431894 -1.8609711 -2.8489157 -1.6793861 -1.9041230
## [8] -0.9609746
```

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)
```

Table 1: Data summary

Name	Galton
Number of rows	928
Number of columns	2
Column type frequency:	
numeric	2
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	

```
#?Galton
```

TO-DO n is the number of observation The column represents the parent and child. The parent is the average of the father and mother height. The child is the numeric vector of the height of child. The data is rounded to the nearest inch. All the heights of female children are multiplied by 1.08 to compensate for sex differences.

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

```
#TO-DO
avg_height = mean(c(Galton$parent, Galton$child))
```

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

```
#TO-DO
n = nrow(Galton)
y = Galton$child
yhat = mean(Galton$child)
```

```
SST = sum((y-yhat)^2)
#only -1 because we are only using one n [videos]
sqrt(SST/(n-1))
```

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

```
#this is the standard deviation of the 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.

```
#TO-DO
mod = lm(child~parent, Galton)
b_0 = coef(mod)[1]
b_1 = coef(mod)[2]
b_0
```

```
## (Intercept)
##    23.94153
```

```
b_1
```

```
##    parent
## 0.6462906
```

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

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

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

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

```
#R squared is very low --> not fitting a lot of variance
#However, it can still work for the data you want. [video] = RMSE
```

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

TO-DO b_0 is the intercept, which means the average mother and father height is 0 and the child is 24 inches tall. b_1 is the slope, the increase of a child's height per increase of the parent's height. R^2 is how well the model accounts the variance of the data so different cases of the data (edge cases) might not be accounted well for in the model. $RMSE$ is the error interval–there is a 9 inch error interval for the prediction if the child's height where you will be right 90% of the time.

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

TO-DO This model is bad because it has a range of 9 inch error. The error margin is too wide to be of useful prediction.

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.

TO-DO Yes because height is part of the parent's genetics. If both parents of the child is 5'7, there is a good chance that the child will become 5'7 because the gene for the 5'7 height is dominant.

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?

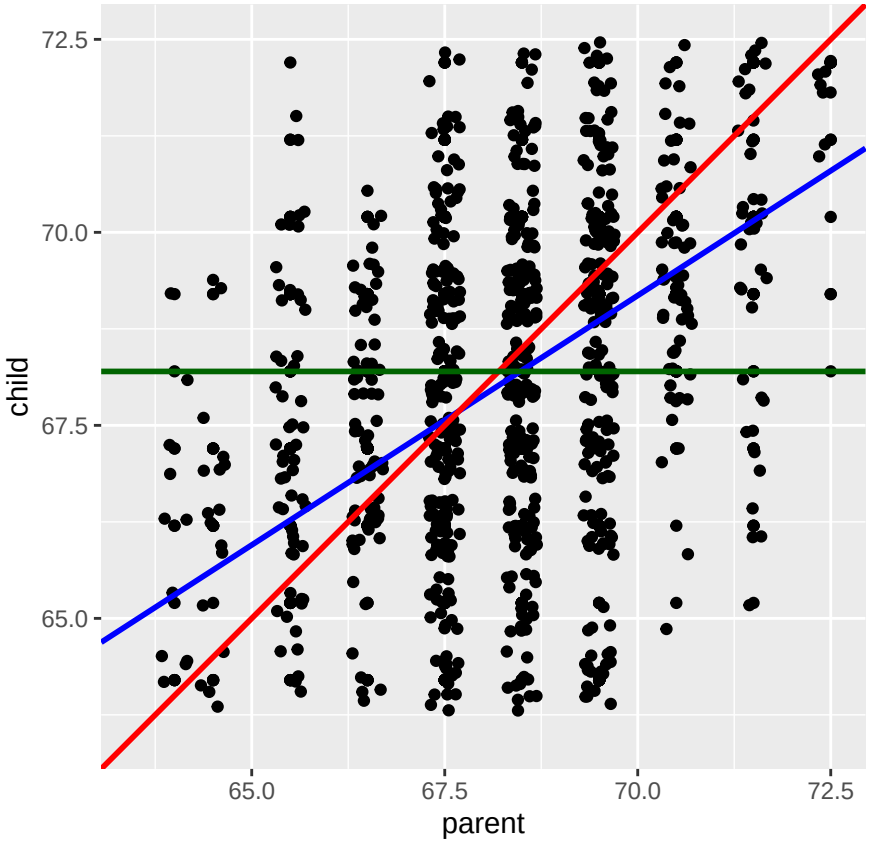
TO-DO β_0 will zero and β_1 is 1 since there will be no increase in height.

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.

```
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 84 rows containing missing values (geom_point).
```



Fill in the following sentence:

TO-DO: Children of short parents became … on average and children of tall parents became … on average.

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"?

If you have short parents, the child will become taller which will bring them closer to the average height. Similarly, if you have tall parents, the children will be shorter to be closer to the average height.

Why should this effect be real?

TO-DO This is so that we don't have people with heights on the opposite spectrum–really short people and really tall people with little to no inbetweens.

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 Galton first called it "Regression towards mediocrity in hereditary stature" so the term "regression" stuck with it ever since. A more descriptive name for building predictive models would be "predictive progression".

You can now clear the workspace. Create a dataset $\mathbb{D}$ which we call `Xy` such that the linear model as R^2 about 50% and RMSE approximately 1.

```
#rm(list = ls())
x = c(6,5,2,4,5,6)
y = c(5,6,3,2,5,7)
Xy = data.frame(x = x, y = y)

simple_ols = my_simple_ols(x,y)

simple_ols$Rsq
```

```
## [1] 0.543552
```

```
simple_ols$RMSE
```

```
## [1] 1.406393
```

Create a dataset $\mathbb{D}$ which we call `Xy` such that the linear model as R^2 about 0% but x, y are clearly associated.

```
x = c(10,30,30,40,50)
y = c(20,15,60,20,25)
Xy = data.frame(x = x, y = y)

simple_ols_ass = my_simple_ols(x,y)
simple_ols_ass$Rsq
```

```
## [1] 0.0003417635
```

Extra credit: create a dataset $\mathbb{D}$ 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
x = c(17,27,27,230,35,59,24)
y = c(1200,2900,270,12500,2000,1999,1000)
Xy = data.frame(x = x, y = y)

simple_ols_arb = my_simple_ols(x,y)
simple_ols_arb$Rsq
```

```
## [1] 0.953173
```

```
simple_ols_arb$RMSE
```

```
## [1] 1000.333
```

Write a function `my_ols` that takes in `X`, a matrix with 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){
  #TO-DO
  n = length(y)
  if (nrow(X) != n){
    stop("x and y needs to be same length.")
  }
  if(class(X[1,]) != "numeric" && class(X[1,]) != "integer"){
    stop("x needs to be numeric.")
  }
  if(class(y) != "numeric" && class(y) != "integer"){
    stop("y needs to be numeric.")
   }
  if( n <= ncol(X)+1){
    stop("n must be more than 2")
  }
  y_bar = sum(y)/n
  b = solve((t(X)%*%X))%*%t(X)*y
  yhat = X%*%b
  e = y - yhat
  SSE = sum(e^2)
  SST = sum((y - y_bar)^2)
  MSE = SSE / (n - 2)
  RMSE = sqrt(MSE)
  Rsq = 1 - (SSE / SST)

  model = list(b = b, df = ncol(X)+1, yhat = yhat, e = e, SSE = SSE, SST = SST, MSE = MSE, RMSE = RMSE,

  class(model) = "my_simple_ols_obj"
```

```
  model
}
```

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 ybar's within group).

```
#TO-DO
cars = MASS::Cars93
lm_mod = lm(Price~Type,cars)
lm_mod
```

```
##
## Call:
## lm(formula = Price ~ Type, data = cars)
##
## Coefficients:
## (Intercept)    TypeLarge  TypeMidsize    TypeSmall   TypeSporty      TypeVan
##     18.2125       6.0875       9.0057      -8.0458       1.1804       0.8875
```

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

```
g = function(x_star, X, y){
  #TO-DO
   b = solve((t(X)%*%X))%*%t(X)*y
   yhat = x_star*b
   yhat
}
```