Lab 3

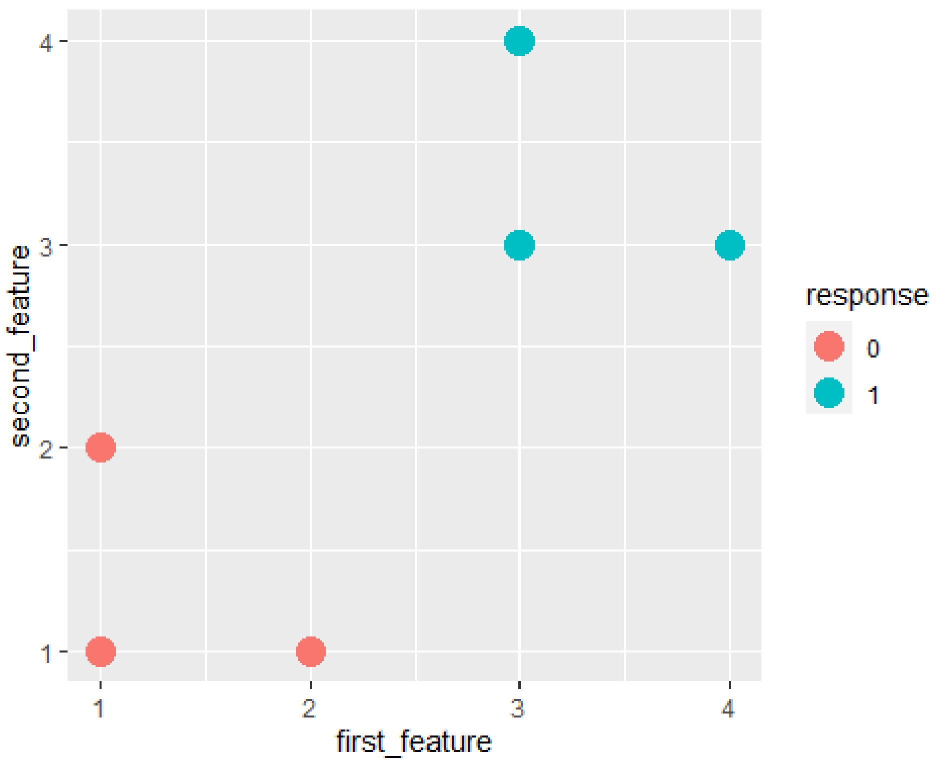
Hubert Majewski

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
```

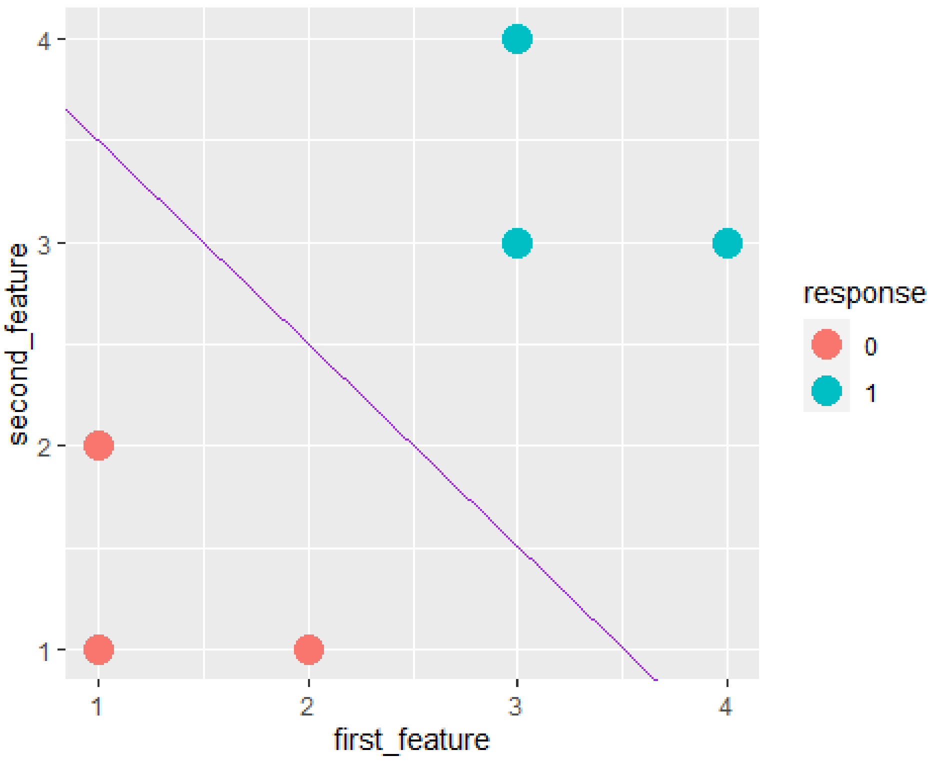


Use the `e1071` package to fit an SVM model to the simple data. Use a formula to create the model, pass in the data frame, set kernel to be `linear` for the linear SVM and don't scale the covariates. Call the model object `svm_model`. Otherwise the remaining code won't work.

```r
pacman::p_load(e1071)

x <- 0;

svm_model = svm(
  Xy_simple,
  formula = Xy_simple$response,
  data = Xy_simple,
  kernel = "linear",
  scale = FALSE
)
```

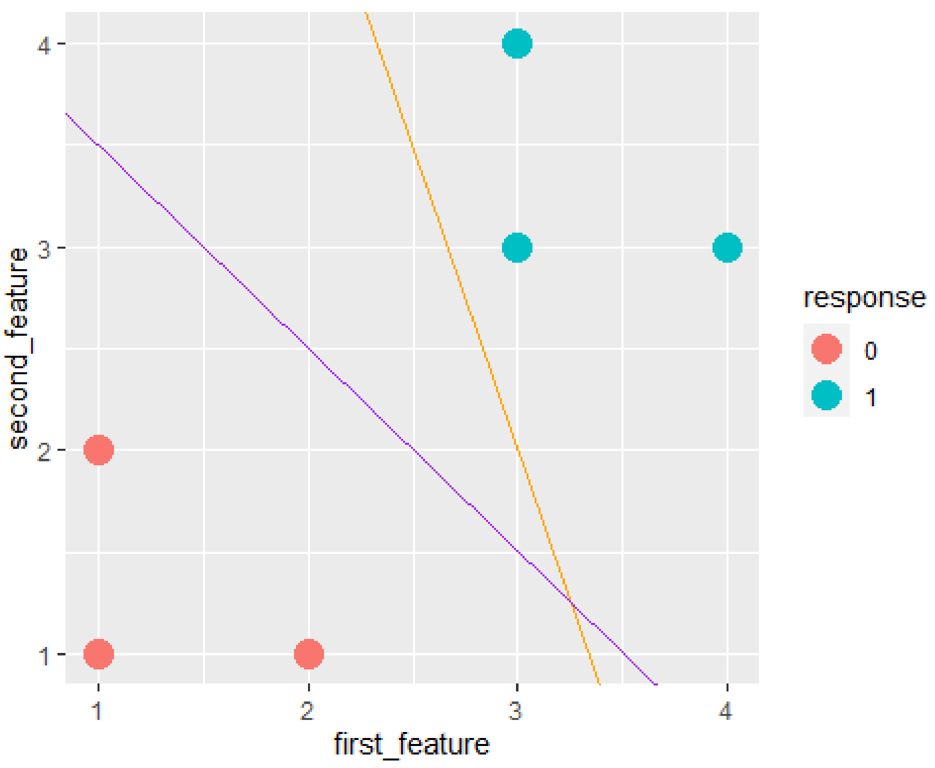
and then use the following code to visualize the line in purple:

```r
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:

```r
#Def from lab 2
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w
= NULL){

    #Add 1 column to matrix making P + 1 size
  Xinput = as.matrix(cbind(1,Xinput))

  z <- rep(0, ncol(Xinput))

  for (i in 1 : MAX_ITER){
    for (j in 1 : nrow(Xinput)) {

        #Get tuple
      x <- Xinput[j, ]

      #Compute y hat
      yHAT <- if(sum(x * z) >= 0) 1 else 0

      #Generate new weights
      for(k in 1:ncol(Xinput)){
        z[k] <- z[k] + (y_binary[j] - yHAT) * x[k]
      }
    }
  }

  return(z)
}


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?

In the previous lab, we have gotten lines with much steeper slopes (orange). For the SVM model, the slopes are much more divisive against the dataset compaired to perceptron.

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){

  #w <- Weighed computed vector

  # for (i < MAX_ITR)
    # fpr (j < n)
        #f for (k < n)
          # If (wj > wi)
            # Move wi to wj
            # Compute the distance with lambda
            # Check/Evaluate the new solution
        # end for
    # end for
    # rank to find the best
  # end for
  # return the result
}
```

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.
#' @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?

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:

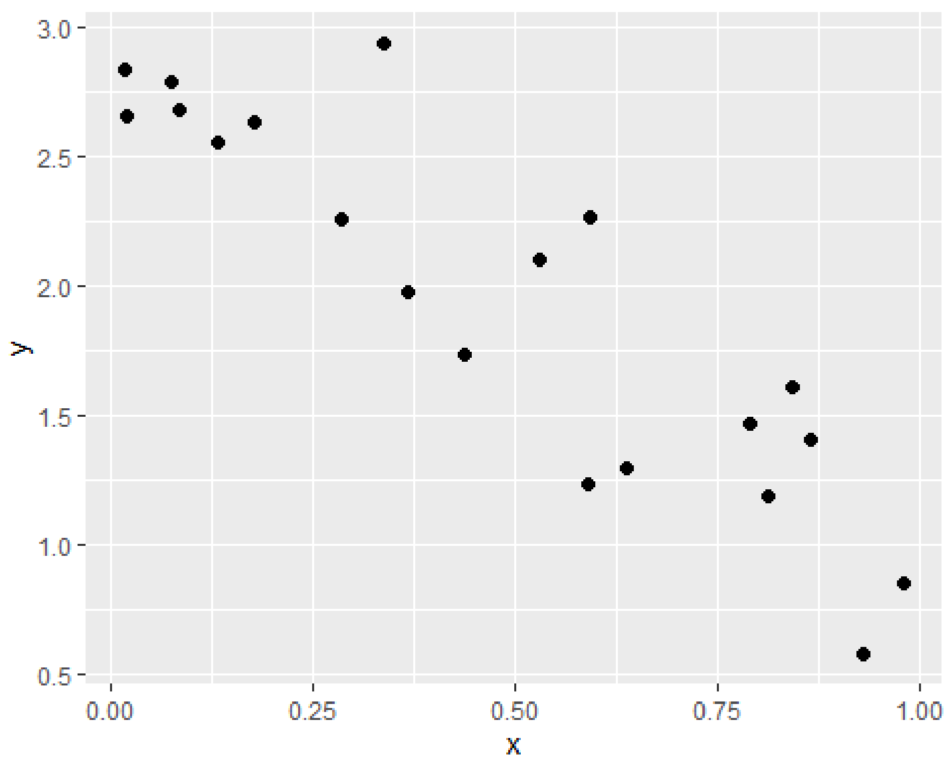
```
n = 20
x = runif(n)
beta_0 = 3
beta_1 = -2
```

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_1*x + beta_0
epsilon = rnorm(n, mean=0, sd=.33)
y = h_star_x + epsilon

h_star_x

##  [1] 2.737619 1.140638 1.812623 1.038574 2.430612 1.940948 2.964265
2.264554
##  [9] 1.315994 2.647056 1.722816 2.831078 1.416647 1.821568 1.267104
2.122651
## [17] 2.958817 2.324622 2.850266 1.371434

epsilon

##  [1] -0.18216970 -0.55918627  0.45690626 -0.18705220 -0.17152791
0.16450676
##  [7] -0.12709758 -0.28526064  0.29265652 -0.01072076 -0.42375787 -
0.15491055
## [13]  0.05701251 -0.58269402  0.14393860 -0.38861686 -0.29956543
0.61326591
## [19] -0.06168687 -0.17852398

y

##  [1] 2.5554494 0.5814513 2.2695291 0.8515218 2.2590844 2.1054547 2.8371677
##  [8] 1.9792937 1.6086502 2.6363356 1.2990584 2.6761672 1.4736600 1.2388740
## [15] 1.4110428 1.7340341 2.6592517 2.9378883 2.7885793 1.1929095
```

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 $beta_1$?

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.

```r
my_simple_ols = function(x, y){
  if (!as.numeric(x) || !as.numeric(y))
    stop("X or Y are not numeric")

  nx <- length(x)
  ny <- length(y)

  if (nx != ny)
    stop("Lengths must be the same")

  if (ny <= 2)
    stop("Length must be greater or equal to 3")


  xBar <- sum(x) / nx
  yBar <- sum(y) / ny
  b1 <- sum(x * y) - nx * xBar * yBar
  b1 <- b1 / (sum(x ^ 2) - nx * xBar ^ 2)

  b0 <- yBar - b1 * xBar

  yHat <- b0 + b1 * x

  e <- y - yHat

  SSE <- sum(e ^ 2)

  SST <- sum( (y - yBar) ^ 2)

  MSE <- SSE / (n - 2)

  RMSE <- sqrt(MSE)

  RSQ <- 1 - SSE / SST

  Model <- list(b_0 = b0, b_1 = b1, yhat = yHat, e = e, SSE = SSE, SST = SST,
MSE = MSE, RMSE = RMSE, Rsq = RSQ)

  #Set using class func
  class(Model) <- "my_simple_ols_obj"

  return(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)
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.

```
mean(my_simple_ols_mod$e)

## [1] 6.106023e-17

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.

```
#prepend 1
x = cbind(1, x)
x

##                  x
##  [1,] 1 0.13119047
##  [2,] 1 0.92968124
##  [3,] 1 0.59368860
##  [4,] 1 0.98071298
##  [5,] 1 0.28469384
##  [6,] 1 0.52952601
##  [7,] 1 0.01786734
##  [8,] 1 0.36772282
##  [9,] 1 0.84200317
## [10,] 1 0.17647182
## [11,] 1 0.63859187
## [12,] 1 0.08446113
## [13,] 1 0.79167626
## [14,] 1 0.58921600
## [15,] 1 0.86644789
## [16,] 1 0.43867453
## [17,] 1 0.02059146
## [18,] 1 0.33768879
## [19,] 1 0.07486691
## [20,] 1 0.81428324
```

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         xx
## 1            1 1 0.13119047
## 2            1 1 0.92968124
## 3            1 1 0.59368860
## 4            1 1 0.98071298
## 5            1 1 0.28469384
## 6            1 1 0.52952601
## 7            1 1 0.01786734
## 8            1 1 0.36772282
## 9            1 1 0.84200317
## 10           1 1 0.17647182
## 11           1 1 0.63859187
## 12           1 1 0.08446113
## 13           1 1 0.79167626
## 14           1 1 0.58921600
## 15           1 1 0.86644789
## 16           1 1 0.43867453
## 17           1 1 0.02059146
## 18              1 1 0.33768879
## 19              1 1 0.07486691
## 20              1 1 0.81428324
## attr(,"assign")
## [1] 0 1 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){

  return(my_simple_ols_obj$b_1 * x_star + my_simple_ols_obj$b_0)

}
```

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), tol= 1e1)
```

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 = 2^(3:20)

errors = 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

  ols <- my_simple_ols(x,y)
  b <- c(ols$b_0, ols$b_1)
  errors[i] = sum((beta - b)^2)
}

errors

##  [1] 1.656868e-02 4.060694e-02 1.217139e-02 9.401264e-03 1.293999e-02
##  [6] 3.354568e-03 2.251536e-03 5.763527e-03 4.255570e-04 5.774405e-04
## [11] 5.583879e-05 5.573904e-05 2.916381e-05 2.140602e-05 3.778777e-05
## [16] 2.698753e-07 1.321370e-06 1.200398e-07

# We see it converges to 0
```

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
_______________________	
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	▁▇▇█▂

We see that the p is 1 as it is the higher measurement and nrow is the number of observations done. We see that the dataset have 928 comparisons between the children and the height of their parents. The mean height for the children was 68.1 and the parent is 68.3. The difference in the means is small because the gnees will carry over to the next generation and so on.

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

```
avgHeight <- mean( c(Galton$parent, Galton$child))
```

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

```
n <- nrow(Galton)

SST <- sum( (Galton$child - mean(Galton$child)) ^ 2)
RMSE <- sqrt(SST / n - 1)

RMSE

## [1] 2.309372
```

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.

```
Model <- lm(child ~ parent, Galton)

b_0 <- coef(Model)[1]
b_1 <- coef(Model)[2]

b_0

## (Intercept)
##    23.94153

b_1

##    parent
## 0.6462906

summary(Model)$sigma

## [1] 2.238547

summary(Model)$r.squared

## [1] 0.2104629

summary(Model)

##
## Call:
## lm(formula = child ~ parent, data = Galton)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.8050 -1.3661  0.0487  1.6339  5.9264
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 23.94153    2.81088   8.517   <2e-16 ***
## parent       0.64629    0.04114  15.711   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.239 on 926 degrees of freedom
## Multiple R-squared:  0.2105, Adjusted R-squared:  0.2096
## F-statistic: 246.8 on 1 and 926 DF,  p-value: < 2.2e-16
```

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

For b_0, we can see that it is impossible to predict when x=0. This is because that there is no parent or child with a height of 0 inches. However, there needs to be a y-intercept for which the small value will fit.

For b_1, we can see that this slope reflects the increase of the parents height. For each increase of the parents height in inches, the child's height grows by the proportional amount.

The RMSE quantity here tells us the error difference between the height values and their mean. We have for example an offset of about 3.8 inches from the mean.

The R^Squared informs us of the proportion of variance in the variables. For example we have 0.21 or 21% of variance portrayed in our model which is small.

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

Given only one measurable feature, this model is good. It can cover over a range of 9 inches up to .96 or 96% of the time. The prediction derived from the model will be confident with at most 12 inches of an error.

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.

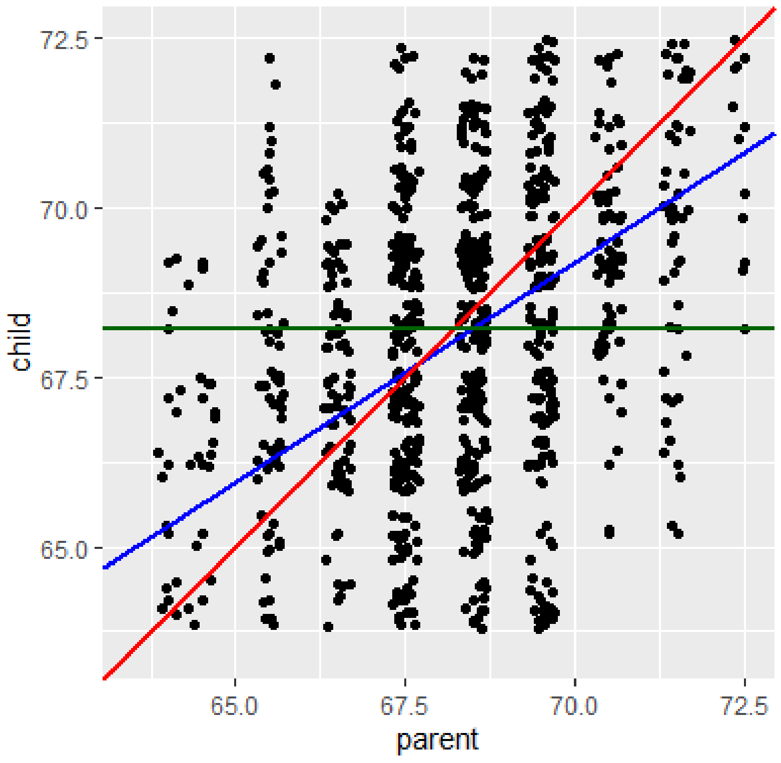
Given that a child inherits the traits of the parent genetically, it makes total sense to assume the parents and children may have the same height. The child may inherit the genes for the height.

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?

We would get a line that is in proportion to the differences in height between the parent and the child. Therefore, our b_0 would be equal to 0 and as the parent will match the height of the child proportionally so b_1 is 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.

```
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 = avgHeight, 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 82 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 short on average.

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

We can see that the relationship between the parent and the children are regressed to the mean of the height. When the parents are taller than the mean, the children were shorter which made them much closer to the mean. This was why Galton called it the "Regression towards mediocrity in hereditary stature"

Why should this effect be real?

As the average height of a human sample population plateaus and remains consistent, the children will balance any deviations/outliers present from 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.

We understand y as a continuous regression be cause we are modeling with continuous variables. Analyzing the performance of a regression will not always result to a mean, this way we can only measure the mean change of a dependent variable. A more descriptive name would be more akin to a least squares linear model where the predictive models with continuous y's to reflect the measuring relationship.

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.

```
x = (2 : 33)
y = log(350, x)
Xy = data.frame(x = x, y = y)

Model = lm(y ~ x)

summary(Model)$r.squared

## [1] 0.5039239

summary(Model)$sigma

## [1] 0.9688295

summary(Model)

##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.7487 -0.5966 -0.2055  0.3109  4.3393
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.31673    0.36702   11.76 9.20e-13 ***
## x           -0.10240    0.01855   -5.52 5.36e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.9688 on 30 degrees of freedom
## Multiple R-squared:  0.5039, Adjusted R-squared:  0.4874
## F-statistic: 30.47 on 1 and 30 DF,  p-value: 5.362e-06
```

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 = (1 : 34)
y = x ^ (100)
Xy = data.frame(x = x, y = y)

Model = lm(y ~ x)

summary(Model)$r.squared

## [1] 0.09448869

summary(Model)

##
## Call:
## lm(formula = y ~ x)
##
## Residuals:
##         Min          1Q      Median          3Q         Max
## -1.481e+152 -9.087e+151 -3.609e+151  2.528e+151  1.239e+153
##
## Coefficients:
##               Estimate  Std. Error t value Pr(>|t|)
## (Intercept) -8.665e+151  8.167e+151  -1.061    0.297
## x            7.439e+150  4.071e+150   1.827    0.077 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.329e+152 on 32 degrees of freedom
## Multiple R-squared:  0.09449,    Adjusted R-squared:  0.06619
## F-statistic: 3.339 on 1 and 32 DF,  p-value: 0.07699
```

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
```

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.

```r
my_ols = function(x, y){

    if (!(is.numeric(x)) && !(is.integer(x)))
      stop("X is not numeric")

    if (!(is.numeric(y)) && !(is.integer(y)))
      stop("y is not numeric")

    ny <- length(y)

    x <- cbind(rep(1, ny), x)

    if (ny != nrow(x))
      stop("Lengths must be the same for multiplication")

    if (ny <= (ncol(x) + 1))
      stop("Length of x must be atleast 1 larger")

    p <- ncol(x)

    df <- ncol(x)

    yBar <- sum(y) / ny

    b <- solve(t(x) %*% x) %*% t(x) %*% y

    yhat <- x %*% b

    e <- y - yhat

    SSE <- sum(t(e) %*% e)

    SST <- sum( (y - yBar) ^ 2)

    MSE <- SSE / (ny - ( p + 1 ))

    RMSE <- sqrt(MSE)

    RSQ <- 1 - (SSE/SST)

    Model <- list(b = b, yhat = yhat, df = df, e = e, SSE = SSE, SST = SST,
MSE = MSE, RMSE = RMSE, Rsq = RSQ, p = p)

    class(Model) <- "my_ols_obj"
    return(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).

```
cars <- MASS::Cars93

Model <- lm(Price ~ Type, data = cars)

ret <- my_ols(as.numeric(data.matrix(data.frame(cars$Type))), cars$Price)
head(ret)
```

```
## $b
##        [,1]
##   22.871020
## x -1.001939
##
## $yhat
##           [,1]
##  [1,] 18.86327
##  [2,] 19.86520
##  [3,] 21.86908
##  [4,] 19.86520
##  [5,] 19.86520
##  [6,] 19.86520
##  [7,] 20.86714
##  [8,] 20.86714
##  [9,] 19.86520
## [10,] 20.86714
## [11,] 19.86520
## [12,] 21.86908
## [13,] 21.86908
## [14,] 17.86133
## [15,] 19.86520
## [16,] 16.85939
## [17,] 16.85939
## [18,] 20.86714
## [19,] 17.86133
## [20,] 20.86714
## [21,] 21.86908
## [22,] 20.86714
## [23,] 18.86327
## [24,] 18.86327
## [25,] 21.86908
## [26,] 16.85939
## [27,] 19.86520
## [28,] 17.86133
## [29,] 18.86327
## [30,] 20.86714
## [31,] 18.86327
## [32,] 18.86327
## [33,] 21.86908
## [34,] 17.86133
## [35,] 17.86133
## [36,] 16.85939
## [37,] 19.86520
## [38,] 20.86714
## [39,] 18.86327
## [40,] 17.86133
## [41,] 17.86133
## [42,] 18.86327
## [43,] 21.86908
## [44,] 18.86327
## [45,] 18.86327
## [46,] 17.86133
## [47,] 19.86520
## [48,] 19.86520
## [49,] 19.86520
## [50,] 19.86520
## [51,] 19.86520
## [52,] 20.86714
## [53,] 18.86327
## [54,] 18.86327
## [55,] 21.86908
## [56,] 16.85939
## [57,] 17.86133
## [58,] 21.86908
## [59,] 19.86520
## [60,] 17.86133
## [61,] 19.86520
## [62,] 18.86327
## [63,] 19.86520
## [64,] 18.86327
## [65,] 21.86908
## [66,] 16.85939
## [67,] 19.86520
## [68,] 21.86908
## [69,] 19.86520
## [70,] 16.85939
## [71,] 20.86714
## [72,] 17.86133
## [73,] 18.86327
## [74,] 21.86908
## [75,] 17.86133
## [76,] 19.86520
## [77,] 20.86714
## [78,] 21.86908
## [79,] 18.86327
## [80,] 18.86327
## [81,] 18.86327
## [82,] 21.86908
## [83,] 18.86327
## [84,] 18.86327
## [85,] 17.86133
## [86,] 19.86520
## [87,] 16.85939
## [88,] 18.86327
## [89,] 16.85939
## [90,] 21.86908
## [91,] 17.86133
## [92,] 21.86908
## [93,] 19.86520
##
## $df
## [1] 2
##
## $e
##               [,1]
##  [1,]  -2.96326531
##  [2,]  14.03479592
##  [3,]   7.23091837
##  [4,]  17.83479592
##  [5,]  10.13479592
##  [6,]  -4.16520408
##  [7,]  -0.06714286
##  [8,]   2.83285714
##  [9,]   6.43479592
## [10,]  13.83285714
## [11,]  20.23479592
## [12,]  -8.46908163
## [13,] -10.46908163
## [14,]  -2.76132653
## [15,]  -3.96520408
## [16,]  -0.55938776
## [17,]  -0.25938776
## [18,]  -2.06714286
## [19,]  20.13867347
## [20,]  -2.46714286
## [21,]  -6.06908163
## [22,]   8.63285714
## [23,]  -9.66326531
## [24,]  -7.56326531
## [25,]  -8.56908163
## [26,]   2.14061224
## [27,]  -4.26520408
## [28,]   7.93867347
## [29,]  -6.66326531
## [30,]  -1.56714286
## [31,] -11.46326531
## [32,]  -8.76326531
## [33,] -10.56908163
## [34,]  -1.96132653
## [35,]  -3.86132653
## [36,]   3.04061224
## [37,]   0.33479592
## [38,]   0.03285714
## [39,] -10.46326531
## [40,]  -5.36132653
## [41,]   1.93867347
## [42,]  -6.76326531
## [43,]  -4.36908163
## [44,] -10.86326531
## [45,]  -8.86326531
## [46,]  -7.86132653
## [47,]  -5.96520408
## [48,]  28.03479592
## [49,]   8.13479592
## [50,]  15.33479592
## [51,]  14.43479592
## [52,]  15.23285714
## [53,] -10.56326531
## [54,]  -7.26326531
## [55,]  -5.36908163
## [56,]   2.24061224
## [57,]  14.63867347
## [58,]  10.03091837
## [59,]  42.03479592
## [60,]  -3.76132653
## [61,]  -4.96520408
## [62,]  -8.56326531
## [63,]   6.23479592
## [64,]  -7.06326531
## [65,]  -6.16908163
## [66,]   2.24061224
## [67,]   1.63479592
## [68,]  -8.36908163
## [69,]  -3.56520408
## [70,]   2.64061224
## [71,]  -0.16714286
## [72,]  -3.46132653
## [73,]  -9.86326531
## [74,] -10.76908163
## [75,]  -0.16132653
## [76,]  -1.36520408
## [77,]   3.53285714
## [78,]   6.83091837
## [79,]  -7.76326531
## [80,] -10.46326531
## [81,]  -7.96326531
## [82,]  -2.36908163
## [83,] -10.26326531
## [84,]  -9.06326531
## [85,]   0.53867347
## [86,]  -1.66520408
## [87,]   5.84061224
## [88,]  -9.76326531
## [89,]   2.84061224
## [90,]  -1.86908163
## [91,]   5.43867347
## [92,]   0.83091837
## [93,]   6.83479592
##
## $SSE
## [1] 8361.872
##
## $SST
## [1] 8584.021
```

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){
  Model <- my_ols(x,y)

  return(c(1, x_star) %*% Model$b)
}

x <- model.matrix( ~Type, cars)[, 2:6]

# Compute prediction
Pred <- g(x[1, ], x, cars$Price)

Pred

##          [,1]
## [1,] 10.16667
```