Lab 3

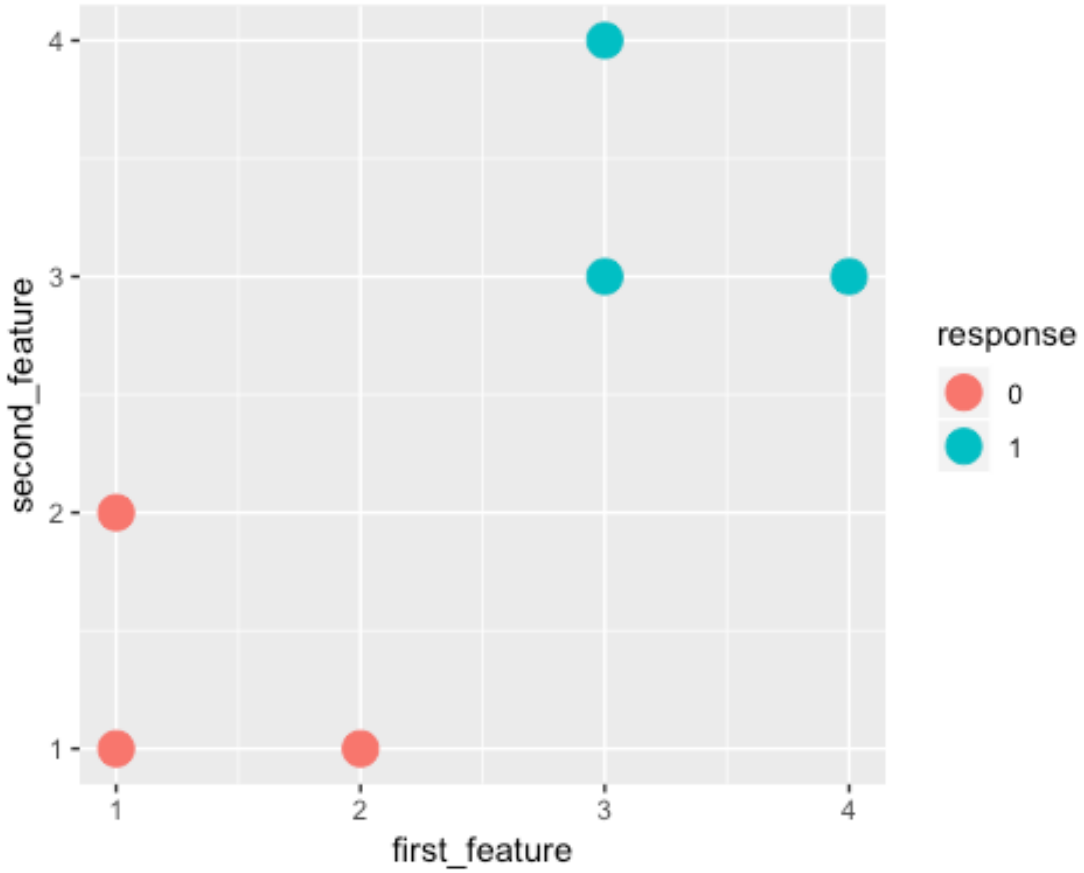
Janine Lim

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
```



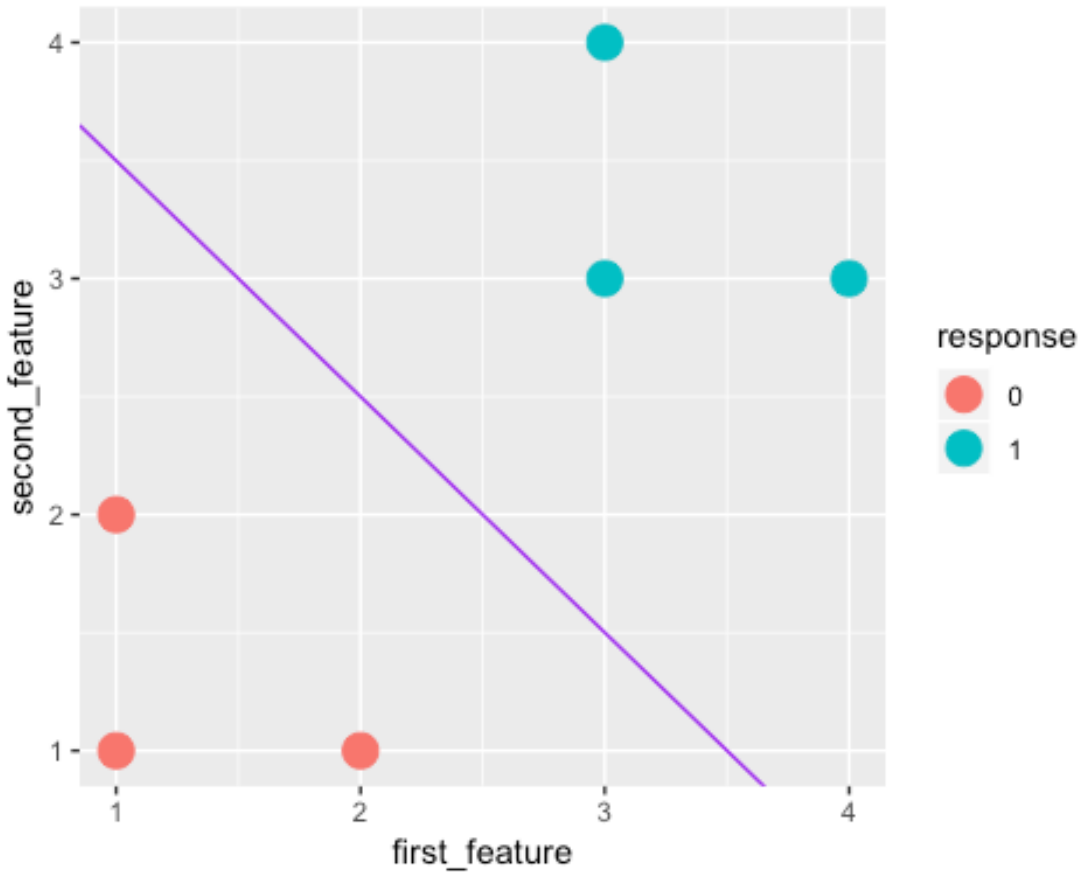
Use the `e1071` package to fit an SVM model to the simple data. Use a formula to create the model, pass in the data frame, set kernel to be `linear` for the linear SVM and don't scale the covariates. Call the model object `svm_model`. Otherwise the remaining code won't work.

```
pacman::p_load(e1071)
svm_model = svm(
  formula = Xy_simple$response~.,
  data = Xy_simple,
  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:

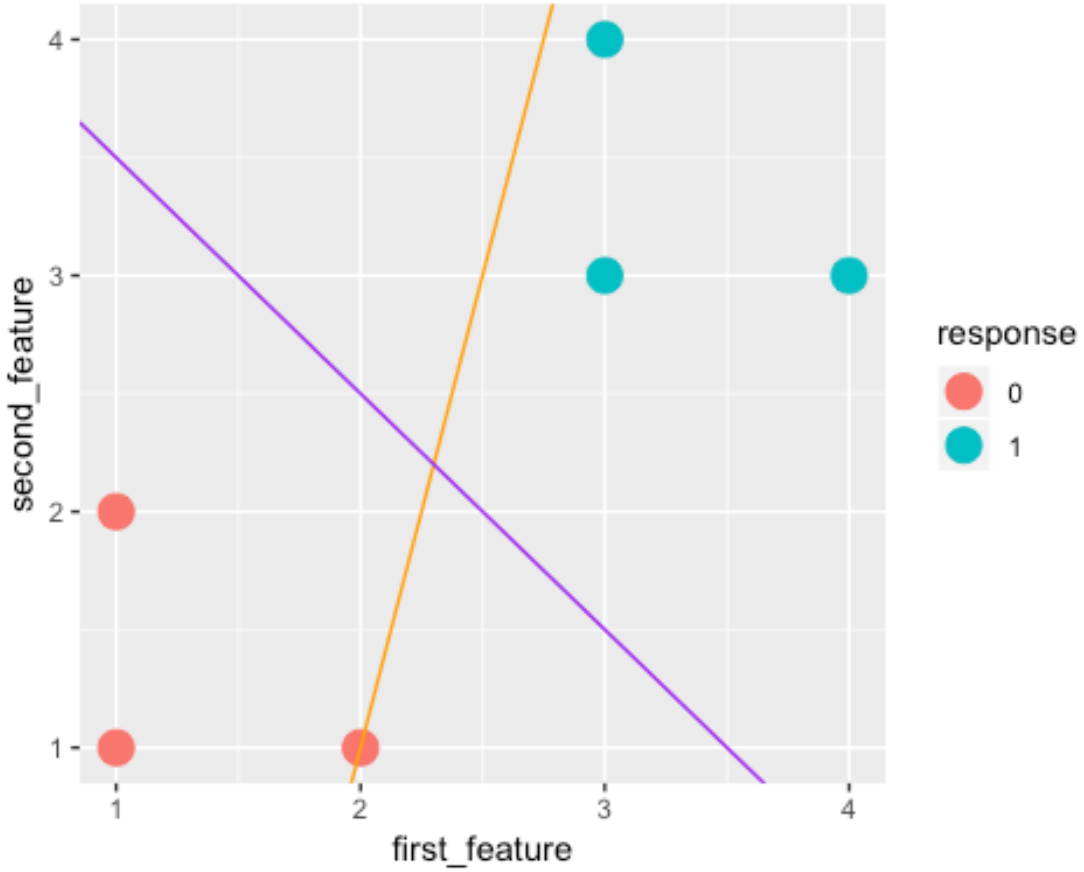
```
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w
= NULL){

Xinput1 = as.matrix(cbind(1,Xinput))
p = ncol(Xinput1)
w = rep(0, p)

for (iter in 1 : MAX_ITER){
  for (i in 1 : nrow(Xinput1)) {
    x_i = Xinput1[i, ]
    yhat_i = ifelse(sum(x_i * w) > 0, 1, 0)
    y_i = y_binary[i]
    for(j in 1:p){
      w[j] = w[j] + (y_i - yhat_i) * x_i[j]
    }
  }
}
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?

Yes, the SVM line is a better fit than the perceptron.

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 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){
  #First we must add a column of 1's to the Xinput matrix with p-columns to
create a matrix that is n x (p+1) and then create a w-vector for our weights
of size p+1.
  #Then we want to iterate through a defined number of iterations to minimize
the misclassifcation error until it equals 0 for all n rows in the X input
matrix.
  #Then we want to set each x_i to be the x_i row from the X input matrix and
similarly, set y_i to be the y_i value from the y_binary input
  # Next we want to create the loss function, SHE (sum of hinge errors) equal
to the sum from 1 to n of maximum of {0, 1/2 - (y_i - 1/2) (w vector times
x_i vector - b)} where b is the intercept
  #Lastly, we want to generate the best line so we take the argmin of the w
vector and b of {1/n SHE + lambda || w vector || ^2}
}
```

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){
  n = nrow(Xinput)
```

```
  Xinput_1 = cbind(1, Xinput)
  w_vec = rep(1, ncol(Xinput_1))
  x_vec = rep(0, n)
  y_vec = rep(0, n)
  h_vec = rep(0,n)
  for (i in 1: MAX_ITER){
    for (j in 1:n){
      x_vec[j] = Xinput_1[j, ]
      y_vec[j] = y_binary[j]
      h_vec[j] = max(0, 0.5 - (y_vec[j]- 0.5)* w_vec*x_vec[j]-b)
    }
    SHE = sum(h_vec)
    which.min(SHE/n + lambda * norm(w_vec)^2)
  }
}
```

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(Xy_simple,
Xy_simple$response)
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?

This should be the same as the e1071 implementation because it is essentially creating the same model as the SVM.

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:

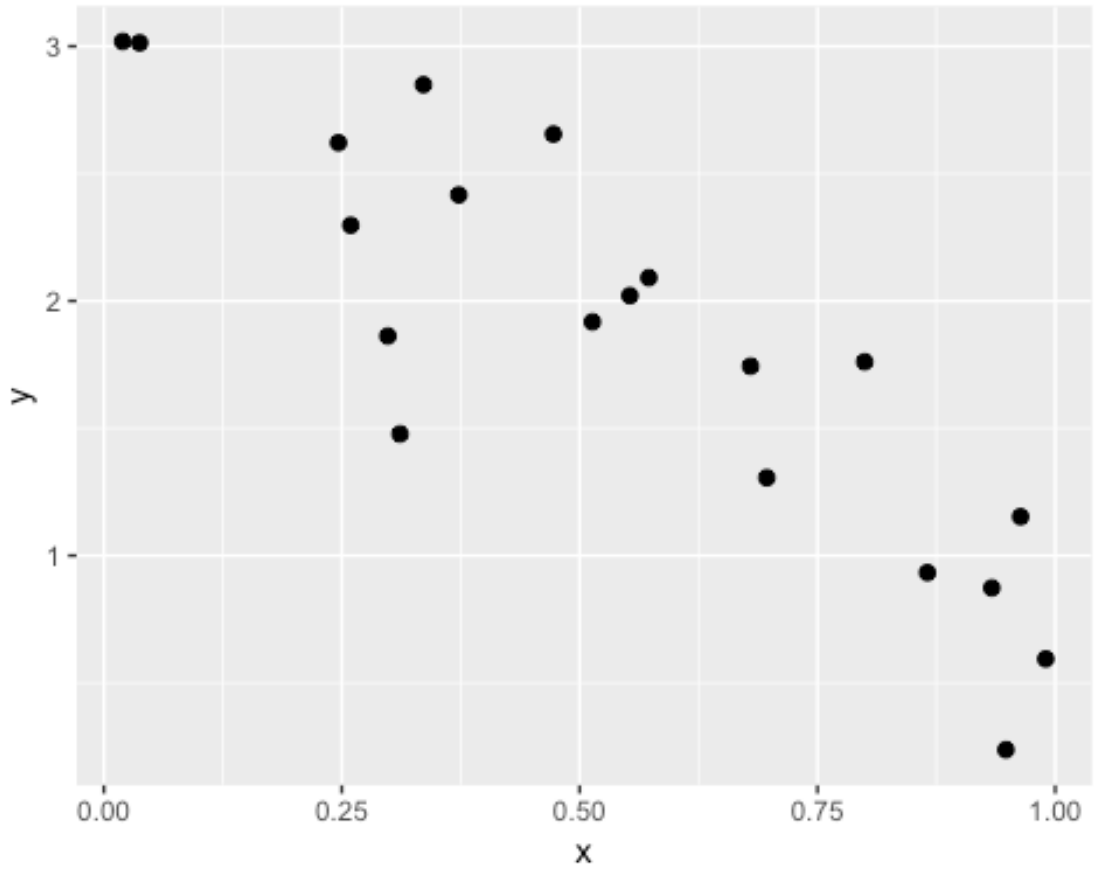
```
n = 20
x = runif(n)
beta_0 = 3
beta_1 = -2
```

Compute $h^*(x)$ as `h_star_x, then draw $\epsilon \sim N(0, 0.33^2)$ asepsilon`, 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 $beta_1$?

Yes - since we know the y-intercept is 3 and slope is 2 and we have random epsilons to offset the error.

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.

```
my_simple_ols = function(x, y){
  n = length(y)

   if (n !=length(x)) {
```

```
    stop("x and y need to be of the 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 greater than 2")
  }
  xbar = sum(x)/n
  ybar = sum(y)/n
  b1 = (sum(x*y)-n * xbar * ybar)/(sum(x^2) - n*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(b0 = b0,
               b1 = b1,
               yhat = yhat,
               e = e,
               SSE = SSE,
               SST = SST,
               MSE = MSE,
               RMSE = RMSE,
               Rsq = Rsq)

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$b0, as.numeric(coef(lm_mod)[1]), tol = 1e-4)
expect_equal(my_simple_ols_mod$b1, 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] 3.330669e-16
```

```
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)
X
```

```
##                   x
##  [1,] 1 0.03750524
##  [2,] 1 0.57279161
##  [3,] 1 0.96351622
##  [4,] 1 0.79967787
##  [5,] 1 0.55286732
##  [6,] 1 0.51343014
##  [7,] 1 0.94814828
##  [8,] 1 0.24650505
##  [9,] 1 0.37306951
## [10,] 1 0.33581682
## [11,] 1 0.99003495
## [12,] 1 0.93328922
## [13,] 1 0.86564335
## [14,] 1 0.29844513
## [15,] 1 0.25938610
## [16,] 1 0.01972272
## [17,] 1 0.67967312
## [18,] 1 0.31128406
## [19,] 1 0.69678494
## [20,] 1 0.47243687
```

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 0.03750524
## 2            1 0.57279161
## 3            1 0.96351622
## 4            1 0.79967787
## 5            1 0.55286732
## 6            1 0.51343014
## 7            1 0.94814828
## 8            1 0.24650505
## 9            1 0.37306951
## 10           1 0.33581682
```

```
## 11           1 0.99003495
## 12           1 0.93328922
## 13           1 0.86564335
## 14           1 0.29844513
## 15           1 0.25938610
## 16           1 0.01972272
## 17           1 0.67967312
## 18           1 0.31128406
## 19           1 0.69678494
## 20           1 0.47243687
## 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$b0 + my_simple_ols_obj$b1*x_star
}
```

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

```
mean(x)
```

```
## [1] 0.5435014
```

```
mean(y)
```

```
## [1] 1.842194
```

```
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)
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$b0, mod$b1)

  error_in_b[i] = sum((beta - b)^2)
}
log(error_in_b, 10)
```

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`.

```
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	▁▃▇▆▂

The Galton data frame is a cross tabulation of 928 children born by their height and the average of their parents' height, using this to predict if parents height is a predictor for

their children's height. We see that the mean height for the parent column is 68.3 and mean height for the child column is 68.1. Here we see that n = 928, and p = 1, the parent's height, used to predict for the children's height.

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 be of this model be?

```
n = nrow(Galton)
SST = sum((Galton$child - mean(Galton$child))^2)
sqrt(SST/(n-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.

```
height = lm(child ~ parent, data = Galton)
b_0 =coef(height)[1]
b_1 = coef(height)[2]
c(b_0, b_1)
```

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

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

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

```
summary(height)$sigma
```

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

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 height for the child when their parents' average height is 0, which we know is impossible and unrealistic. b_0 is measured in inches. b_1 states that for every 1 inch increase in the average of the parents' height, the child's height will increase by 0.6462906. b_1 is measured in inches. Since we have an RMSE of approximately 2.24, this is saying that we have 95% of the true heights of the children within plus 4.48 + mean of the child column and minus 4.48 of the child column. RMSE is also measured in inches. Since we

have an R^2 value of approximately 0.21, there is an approximately 21% change in variance between our model and the null model, which would be the average of the child column.

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

This model is pretty good for only having one input feature to predict on height for the children, but we still have quite a error of 9 inches. We can also note that this model is pretty good for predicting with this population from the 1800's but may perform worse now.

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.

Yes, this is reasonable because children have their parents' genes and so would inherit a similar height to their parents.

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?

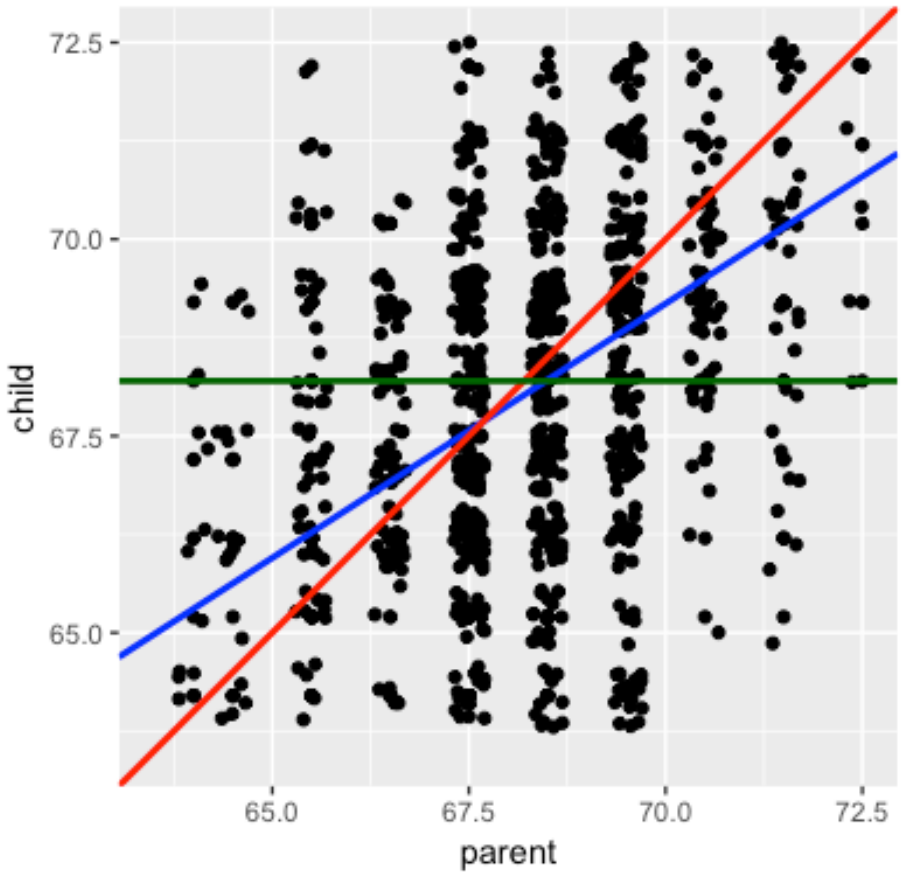
b_0 would be 0 and b_1 would be 1 to denote the line of y = x, since we are assuming they have the same height.

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



Fill in the following sentence:

Children of short parents became tall 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”?

Galton called it regression towards mediocrity in hereditary stature because he noticed that height of children with taller parents were short, and the height of children with short parents are tall, which balances the average height of the children who are tall and short.

Why should this effect be real? This effect should be real because the average height will remain around the same since there is a balancing act between tall parents and short parents, and so even if there is a combination of one tall parent and one short parent, there will be some height compensation with their children.

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.

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

```
x = c(1.5, 2.5, 2, 0)
y = c(2, 0, 3, 3.5)
Xy = data.frame(x = x, y = y)
mod = lm(y~x, data =Xy)
summary(mod)$r.squared
```

```
## [1] 0.5590062
```

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

```
## [1] 1.258897
```

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 = 1:100
y = x^25 + x^75
Xy = data.frame(x = x, y = y)
mod = lm(y~x, data = Xy)
summary(mod)$r.squared
```

```
## [1] 0.08021842
```

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
```

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){
  n = length(y)
  p = ncol(X)
  if (!is.numeric(X) && !is.integer(X)) {
    stop("X must be numeric")
  }
  X = as.matrix(cbind(1, X))

  if (n != nrow(X)){
    stop("X and y need to be the same length.")
  }
```

```
  if(class(y) !='numeric' && class(y) !='integer'){
    stop("y needs to be numeric.")
   }
  if(n<=ncol(X)+1){
    stop("n must be greater than 2.")
  }

  y_bar = sum(y)/n

  b= solve(t(X) %*% X) %*% t(X) %*% y
  y_hat = X %*% b

  df = ncol(X)
  e = y - y_hat
  SSE = sum(t(e) %*% e)
  SST = sum((y - y_bar)^2)
  MSE = SSE / (n-(ncol(X)+1))
  RMSE = sqrt(MSE)
  Rsq = 1 - (SSE/SST)

  model = list(b=b,
               y_hat = y_hat,
               df = df,
               e=e,
               SSE = SSE,
               SST = SST,
               MSE = MSE,
               RMSE = RMSE,
               Rsq = Rsq)
  class(model) = "my_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).

```
cars = MASS::Cars93
mod = lm(Price~Type, data=cars)
values = my_ols(as.numeric(data.matrix(data.frame((cars$Type)))), cars$Price)
```

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){
  b = my_ols(X,y)$b
  x_star = c(1,x_star)
  x_star %*% b
}
```

```r
X = model.matrix(~Type, cars)[,-1]
g(X[1, ], X, cars$Price) #making sure they match

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

predict(mod, cars[1,]) #making sure they match

##        1
## 10.16667
```