Lab 3

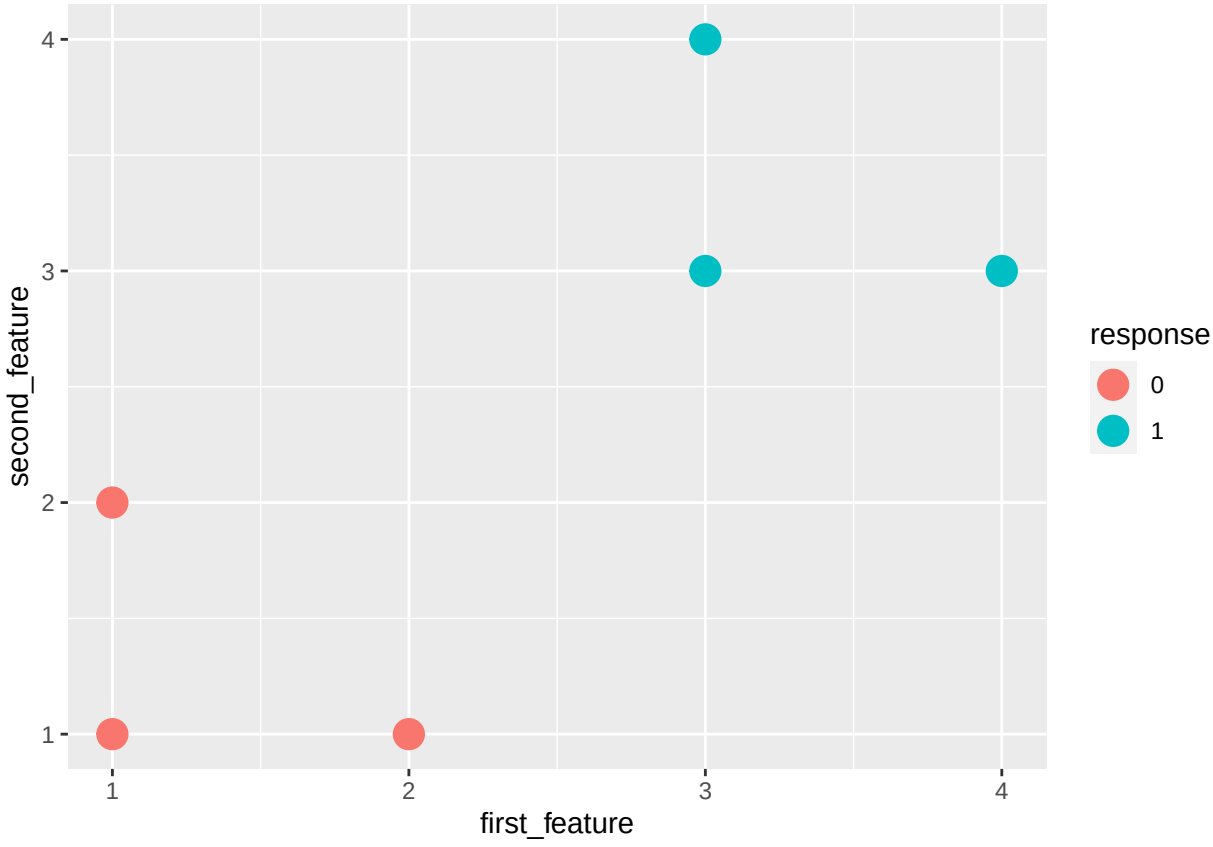
Brendan Gubbins

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
```

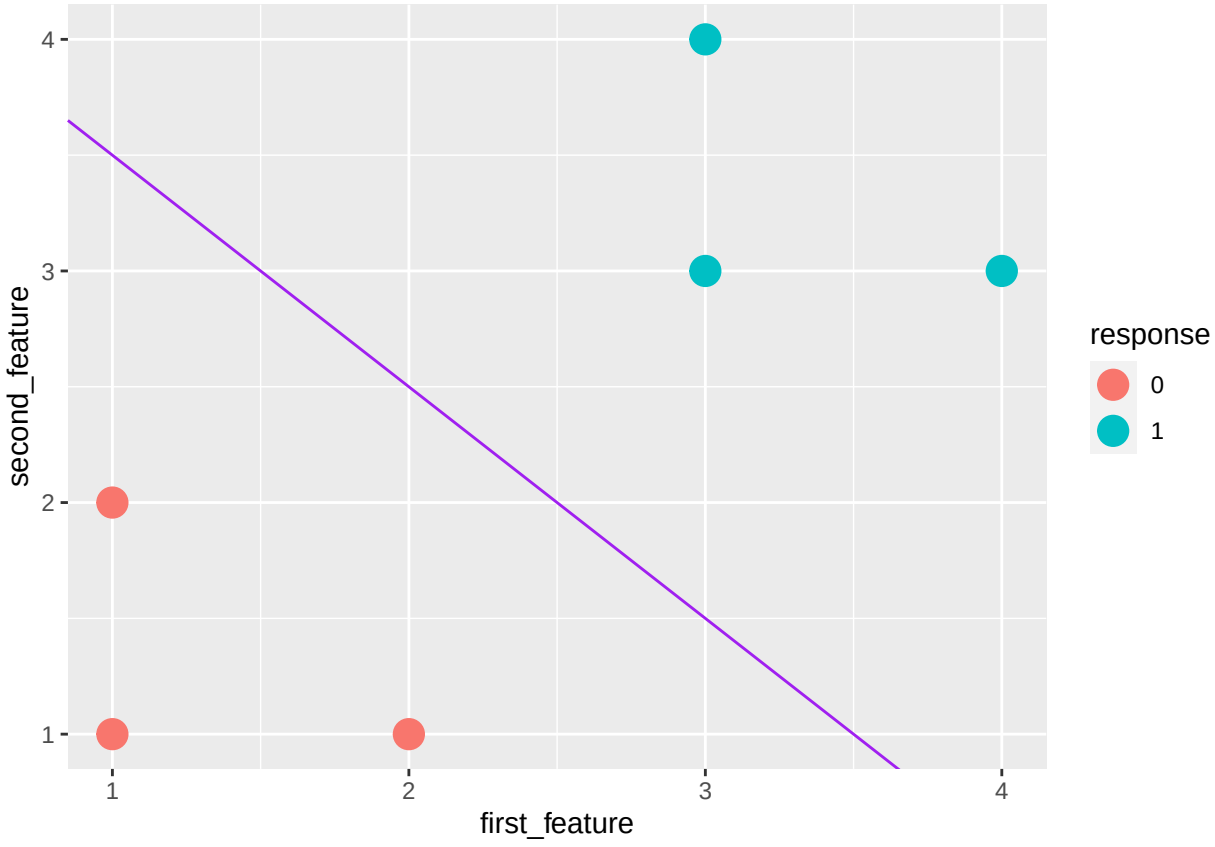


Use the `e1071` package to fit an SVM model to the simple data. Use a formula to create the model, pass in the data frame, set kernel to be `linear` for the linear SVM and don't scale the covariates. Call the model object `svm_model`. Otherwise the remaining code won't work.

```
pacman::p_load(e1071)
Xy_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
svm_model = svm(
  x = Xy_simple_feature_matrix, # not sure why I needed this to remove error
  formula = Xy_simple_feature_matrix,
  data = Xy_simple$response,
  kernel = "linear",
  scale = FALSE
)
#svm_model = svm(Xy_simple_feature_matrix, Xy_simple$response, kernel = "linear", cost = (2 * n * lambd
```

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:

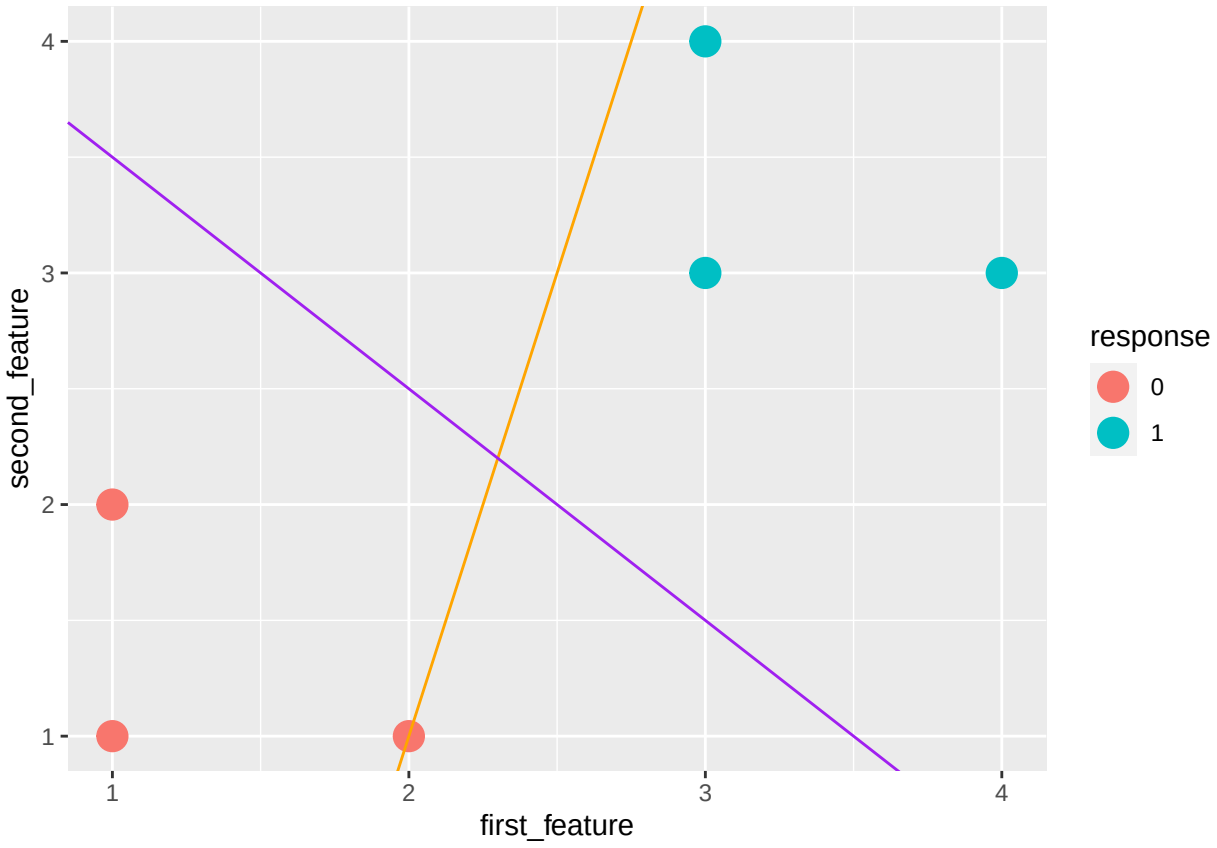
```
# sourcing my perceptron function from lab02

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

  Xinput = as.matrix(cbind(1, Xinput[,, drop = FALSE]))
  n = nrow(Xinput)
  p = ncol(Xinput)
  w = rep(0, p)
  for (i in 1 : MAX_ITER) {
    for (j in 1 : n) {
      x_j = Xinput[j,]
      y_hat = ifelse(sum(x_j * w) > 0, 1, 0)
      for (k in 1 : p) {
        w[k] = w[k] + (y_binary[j] - y_hat) * 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?

The purple SVM line is clearly far superior to the perceptron, because it cleanly splits the data. The orange perceptron line is touching the data at (2,1).

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){
  # for 1 to MAX_ITER
  # initialize w and SHE variables
  #   for 1 to nrow(Xinput)
  #     set w to satisfy such that ||w|| satisfies (y_binary_i - 1/2)(w * Xinput[i,] - b) >= 1/2
  #     min_w = min(w, min_w)
  #   compute sum of the hinge errors (SHE) by summing max(0, 1/2 - (y_binary_i - 1/2)(w * Xinput[i,] -
```

```
  #   optimize for argmin w,b (1/nrow(Xinput) * SHE + lambda * ||w||^2)
  # return 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
#'
#' @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?

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:

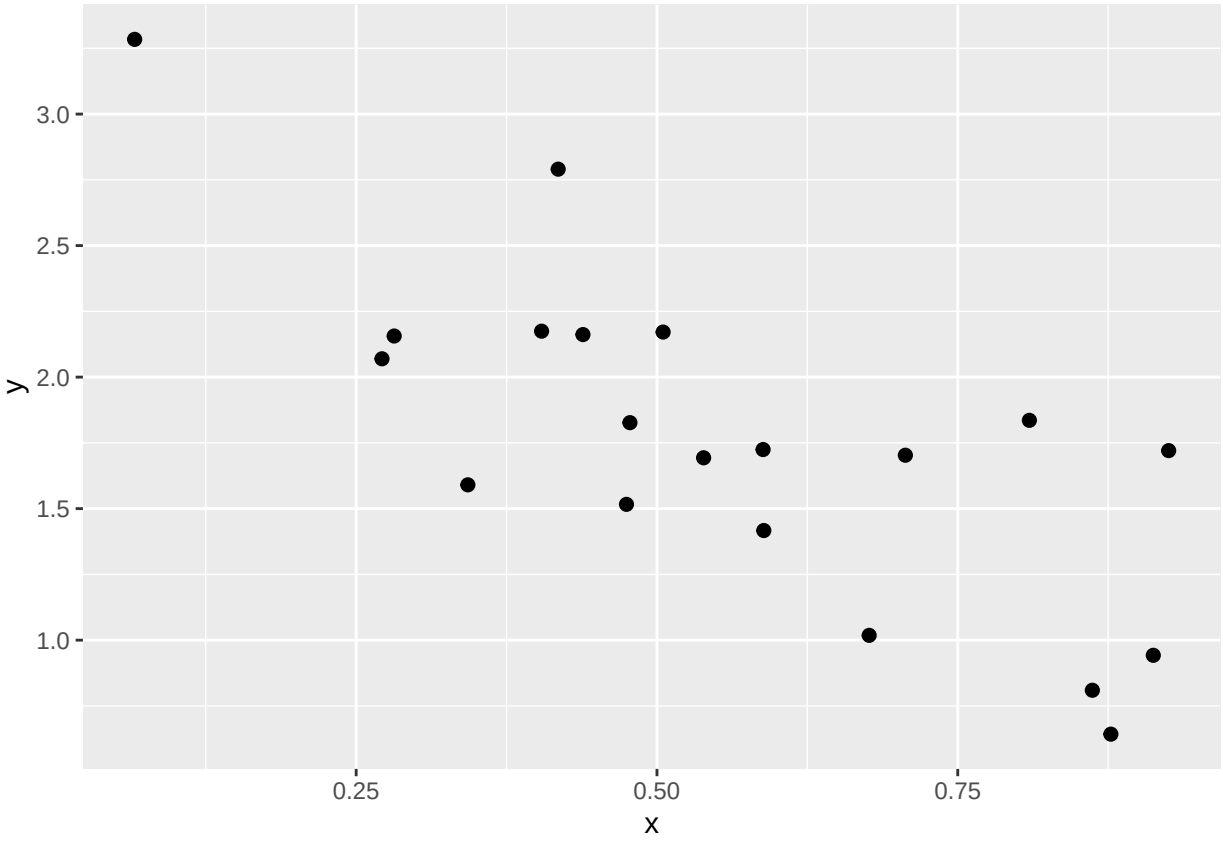
```
n = 20
x = runif(n)
beta_0 = 3
beta_1 = -2
```

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 β_0 and β_1?

beta_0 is 3 and beta_1 is -2. The graph makes sense because if you fit a line on these data points it will be about a -2 slope (beta_1) and the intercept is about 3 (beta_0)

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 must be same length.")
  }

  if (class(x) != 'numeric' & class(y) != 'integer') {
    stop("x needs to be numeric.")
  }


  if (class(y) != 'numeric' & class(x) != '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 # return
}
```

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.

```
residuals = mean(my_simple_ols_mod$e)
# expect_equal(residuals, 0) # not good in real world, residuals may go to 0 on R
expect_equal(residuals, 0, tol = 1e-4)
```

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

```
X = cbind(1, x) # cbind makes a matrix
X
```

```
##                   x
##  [1,] 1 0.70646336
##  [2,] 1 0.91253894
##  [3,] 1 0.80953563
##  [4,] 1 0.53876750
##  [5,] 1 0.92531829
##  [6,] 1 0.40411253
##  [7,] 1 0.41790047
##  [8,] 1 0.86188017
```

```
##  [9,] 1 0.87728114
## [10,] 1 0.34281368
## [11,] 1 0.50510502
## [12,] 1 0.06587188
## [13,] 1 0.47754040
## [14,] 1 0.43847726
## [15,] 1 0.27145888
## [16,] 1 0.58819483
## [17,] 1 0.28158803
## [18,] 1 0.58879457
## [19,] 1 0.47466049
## [20,] 1 0.67637947
```

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 0.70646336
## 2            1 0.91253894
## 3            1 0.80953563
## 4            1 0.53876750
## 5            1 0.92531829
## 6            1 0.40411253
## 7            1 0.41790047
## 8            1 0.86188017
## 9            1 0.87728114
## 10           1 0.34281368
## 11           1 0.50510502
## 12           1 0.06587188
## 13           1 0.47754040
## 14           1 0.43847726
## 15           1 0.27145888
## 16           1 0.58819483
## 17           1 0.28158803
## 18           1 0.58879457
## 19           1 0.47466049
## 20           1 0.67637947
## 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){
  y_star = my_simple_ols_obj$b_0 + my_simple_ols_obj$b_1 * x_star
  y_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)
ns = 10^(1:8)
error_in_b = array(NA, length(ns)) # errors in b not being beta
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)
}
log(error_in_b, 10)
```

```
## [1] -0.7330785 -2.6070310 -3.7830333 -3.5786532 -4.1990968 -7.4610560 -8.1076123
## [8] -9.3463917
```

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

Table 1: Data summary

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 number of rows n = 928, so there are 928 parents heights averaged, as well as a corresponding child height. The two columns represent parents and child, p = 2. The mean averaged height of parents is 68.3 inches, and the mean height of children is 68.1 inches. The lowest height found in the 0th percentile is 61.7 inches (child), and the largest height found in the 100th percentile is 73.7 inches (child). Based on the histogram, the vast majority of height is found in the 25th-75th percentile. As described in the dataset, all female children had their heights multiplied by 1.08 as a balancing effect.

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

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

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

```
n = nrow(Galton)
SST = sum((Galton$child - mean(Galton$child))^2)
sqrt(SST/(n-1)) # not n-2, only fitting one ybar
```

```
## [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)
coef(mod)
```

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

```
b_0 = coef(mod)[1]
b_1 = coef(mod)[2]
summary(mod)$sigma # +- 2.23, +- 4.5 95% of the time
```

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

```
summary(mod)$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 in inches of the child's height. In this case, if the average of father and mother heights are 0 (absurd), the child will be 23.9~ inches tall. b_1 is the increase in average child height per 1 increase in average height of father and mother (child gets 0.65~ inches taller per 1 inch of average parent height). The RMSE is 2.23~, meaning that 95% of the time, there is a +- of 4.46~ inches on the child's predicted height. R^2 is 21% which is fairly low, this means that the model does not fit the variance well.

b_0 intercept/if parent height is 0, child height is 23.94 inch. b_1 increase in child (.64) height per average height of parent. RMSE tells the range. R^2 is low (21% of the variance explained), so the model is not accounting for the data variance.

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

It's a pretty bad model because from the RMSE, you can tell somebody's height 95% of the time from a +- of about 4.46 inches. This is not good because being off by 4.46 inches in height 95% of the time is significant (a range of between 5'4" to 6'+).

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.

It's reasonable for the most part to assume that children will be the same height as their parents i.e taller parents will produce taller children and shorter parents will produce shorter children on average. It would be unreasonable to assume that children height is randomized, because they are inheriting DNA from 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?

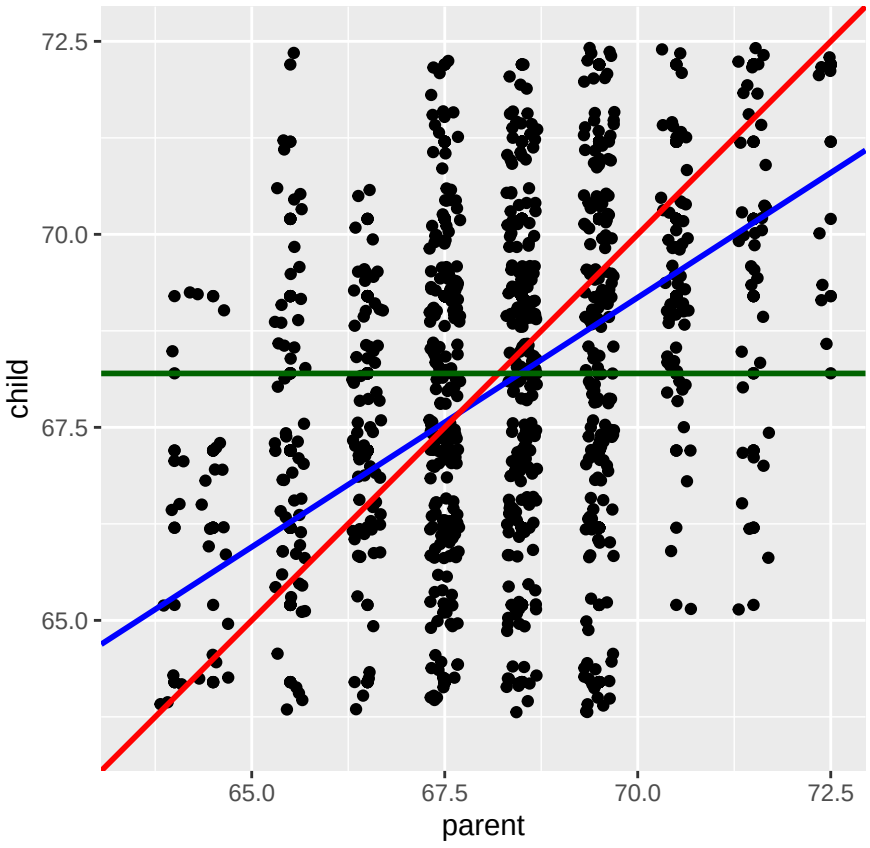
beta_0 would be the intercept at 0 and beta_1 would be 1. This means there would be a 1:1 change when determining child height based on parent height. Any parent height produces the same exact child height.

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



Fill in the following sentence:

Children of short parents became taller than expected on average and children of tall parents became shorter than expected on average.

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

Because the children of very tall parents were shorter than expected, and therefore closer to the mean height, and the children of very short parents were taller than expected, closer to the mean height. These children of very tall parents/very short parents "regressed" towards the mean height.

Why should this effect be real?

This effect should be real because short parents would keep producing smaller and smaller children with no limit, and tall parents would keep producing taller and taller children with no end. This effect keeps humans generally of similar size on average.

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.

It's called regression because of Galton's coinage of "Regression" to the mean. A more appropriate description/name for predictive models could be called "historical sampling," which takes historical data as samples and spits out predictions based on that data.

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(7.5, 11, 9, 9, 1)
y = c(2, 5, 3, 5, 2)
Xy = data.frame(x = x, y = y)
model = lm(y ~ x)
summary(model)$r.squared
```

```
## [1] 0.501566
```

```
summary(model)$sigma
```

```
## [1] 1.236338
```

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(1,4,8,12)
y = c(4,1,2,3)
Xy = data.frame(x = x, y = y)
model = lm(y ~ x)
summary(model)$r.squared
```

```
## [1] 0.01818182
```

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
x = c(5,90,30,15,2,50,44,1020,999)
y = c(50,1000,700,899,2000,11,2,10000,7802)
Xy = data.frame(x = x, y = y)
model = lm(y ~ x)
summary(model)$r.squared
```

```
## [1] 0.9449758
```

```
summary(model)$sigma
```

```
## [1] 934.52
```

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 (n != nrow(X)) {
    stop("X # rows and y must be same length.")
  }

  if (class(X[1,]) != 'numeric' & class(y) != 'integer') {
    stop("X needs to be numeric.")
  }

  if (class(y) != 'numeric' & class(X[1,]) != 'integer') {
    stop("y needs to be numeric.")
  }

  if (n <= p + 1) {
    stop("n must be more than p+1.")
  }

  df = p + 1
  Xt = t(X)
  XtX = Xt %*% X
  XtXinv = solve(XtX)
  b = XtXinv %*% Xt %*% y
  y_bar = sum(y)/n
  yhat = X %*% b
  e = y - yhat
  SSE = t(e) %*% e
  SST = sum((y - y_bar)^2)
  MSE = SSE /(n - (p + 1))
  RMSE = sqrt(MSE)
  Rsq = 1 - SSE / SST

  model = list(b = b, yhat = yhat, e = e, df = df, SSE = SSE, SST = SST, MSE = MSE, RMSE = RMSE, Rsq = R
  class(model) = "my_ols"
  model # return
}
```

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
lm(Price ~ Type, cars)
```

```
##
## 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){
  b = solve(t(X) %*% X) %*% t(X) %*% y
  y_hat = x_star %*% b
  y_hat
}
```