Lab 3

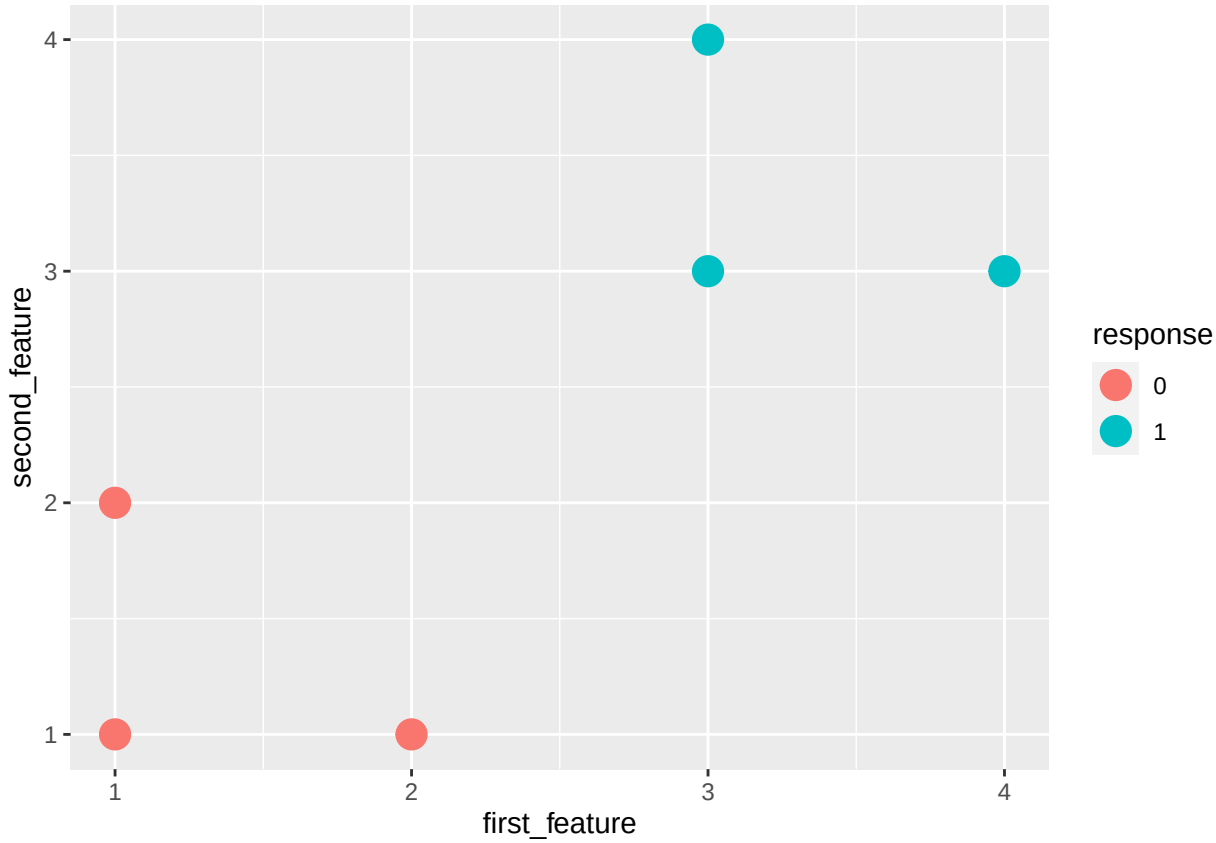
Frank Palma Gomez

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
```



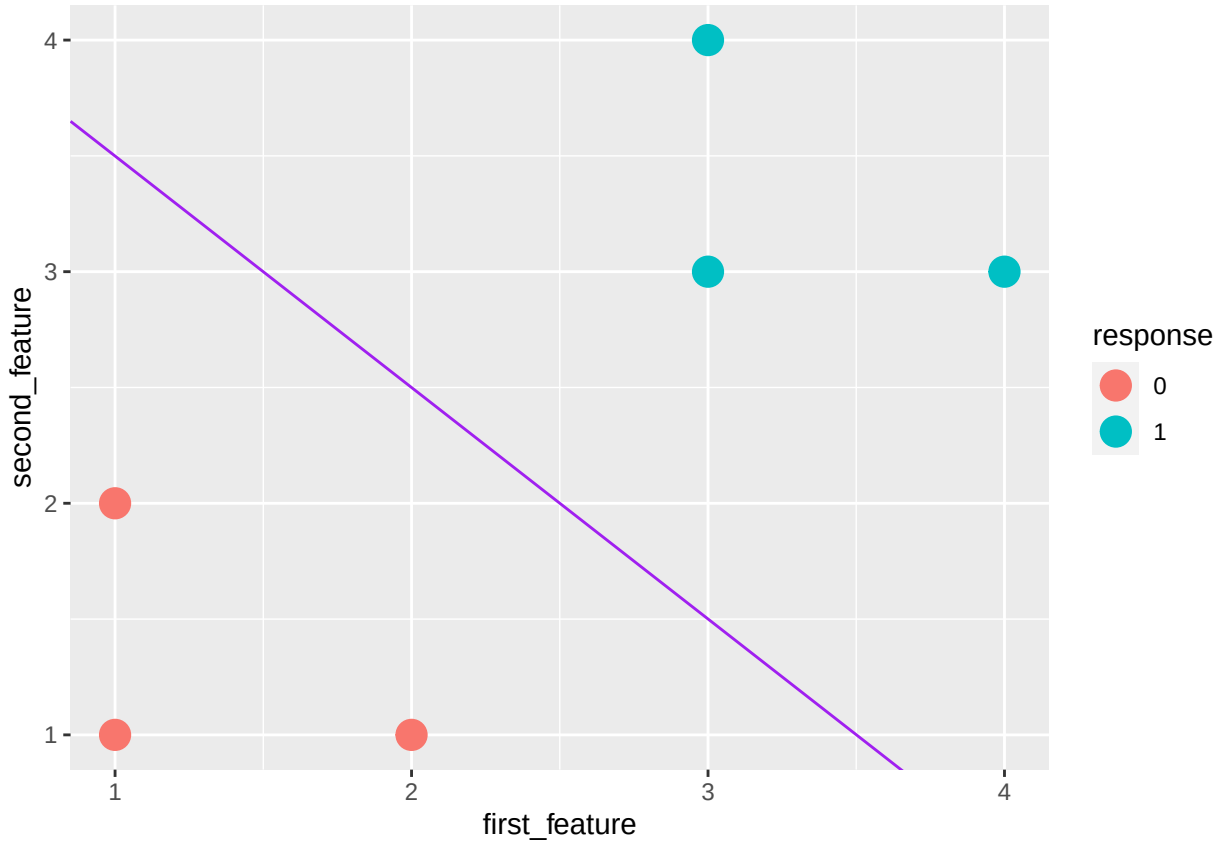
Use the `e1071` package to fit an SVM model to the simple data. Use a formula to create the model, pass in the data frame, set kernel to be `linear` for the linear SVM and don't scale the covariates. Call the model

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, ] #
)
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
```

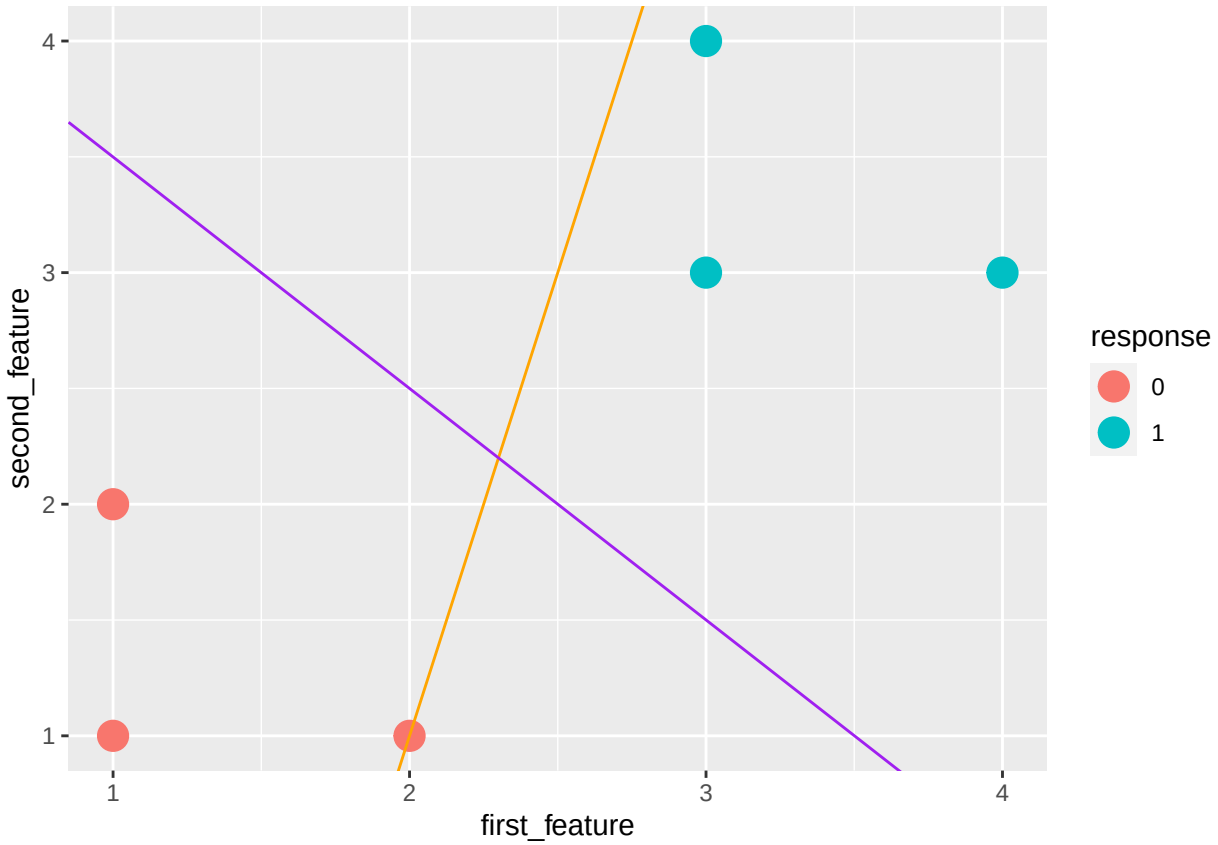


Source the `perceptron_learning_algorithm` function from lab 2. Then run the following to fit the perceptron and plot its line in orange with the SVM's line:

```
#' Perceptron with p features
#'
#' Implementation of the Perceptron model with p features.
```

```r
#'
#' @param Xinput      A matrix of features for training data observations
#' @param y_binary    Vector of training data labels
#' @param MAX_ITER    Number of iterations the models must do to converge
#' @param w           Weight vector used to learn the parameters
#'
#' @return            The computed final parameter (weight) as a vector of length p + 1
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w = NULL) {
  p = ncol(Xinput) # get the number of features
  if (is.null(w)) {
    # check if w is NULL
    w = rep(0, p + 1) # p + 1 since we include the bias
  } else if (length(w) != p + 1) {
    stop("The length of w must be the nrow(Xinput) + 1")
  }

  X_with_bias = as.matrix(cbind(1, Xinput))
  n = nrow(X_with_bias)

  for (iter in 1 : MAX_ITER) {
    for (i in 1 : n) {
      x_i = X_with_bias[i, ] # get the ith observation
      yhat = ifelse(sum(x_i * w) > 0, 1, 0)  # predicted label
      y_i = y_binary[i] # true label
      # update weights and biases
      for (j  in 1:ncol(X_with_bias)) {
        w[j] = w[j] + (y_i - yhat) * x_i[j]
      }
    }
  }

  w # return weight vector
}
```

```r
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 SVM line is better because it maximizes the space between the support vectors. The perceptron is able to divide the classes but there still one observation that lands on top of the separating line.

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.

```
pacman::p_load(optimx)
```

```
#' 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) {
  # check that Xinput and y_binary share the same number of rows

  # initialize a weight vector with dimensions p + 1. Where p is the number of cols in Xinput
  # with the bias term at the first position
```

```
  # initialize the sum of hinge loss function

  # iterate through Xinput and y_binary in the row direction
  # calculate the  sum of hinge loss
  # find the argmin of (1 / n) * sum_hinrors + lge_erambda * norm_vec(w)
  # you can do this by taking the gradient w.r.t w and setting it equal to zero
  # update the weight matrix with an optimizer

  # repeat the previous MAX_ITER times


  # return the weight vector
}
```

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){

  n = nrow(Xinput)
  p = ncol(Xinput)

  if (nrow(Xinput) != nrow(y_binary)) {
    stop("X and y must have the same number of rows")
  }
  # check that X is numeric and integer
  if (class(Xinput[, 1])  != "numeric" & class(Xinput[, 1]) != "integer") {
    stop("x needs to be numeric")
  }
  # check that y is numeric and integer
  if (class(y_binary[1, ])  != "numeric" & class(y_binary[1, ]) != "integer") {
    stop("y needs to be numeric")
  }
  # check that there are more than 2 rows
  if (n <= 2) {
    stop("Dimensions of n must be greater than 2")
  }

  # first element is the bias
  w = rep(1, p+1)

  cost_fn = function(w) {
      norm_vec = function(x) sqrt(sum(x^2))
      y_part = as.numeric(y_binary - 0.5)
```

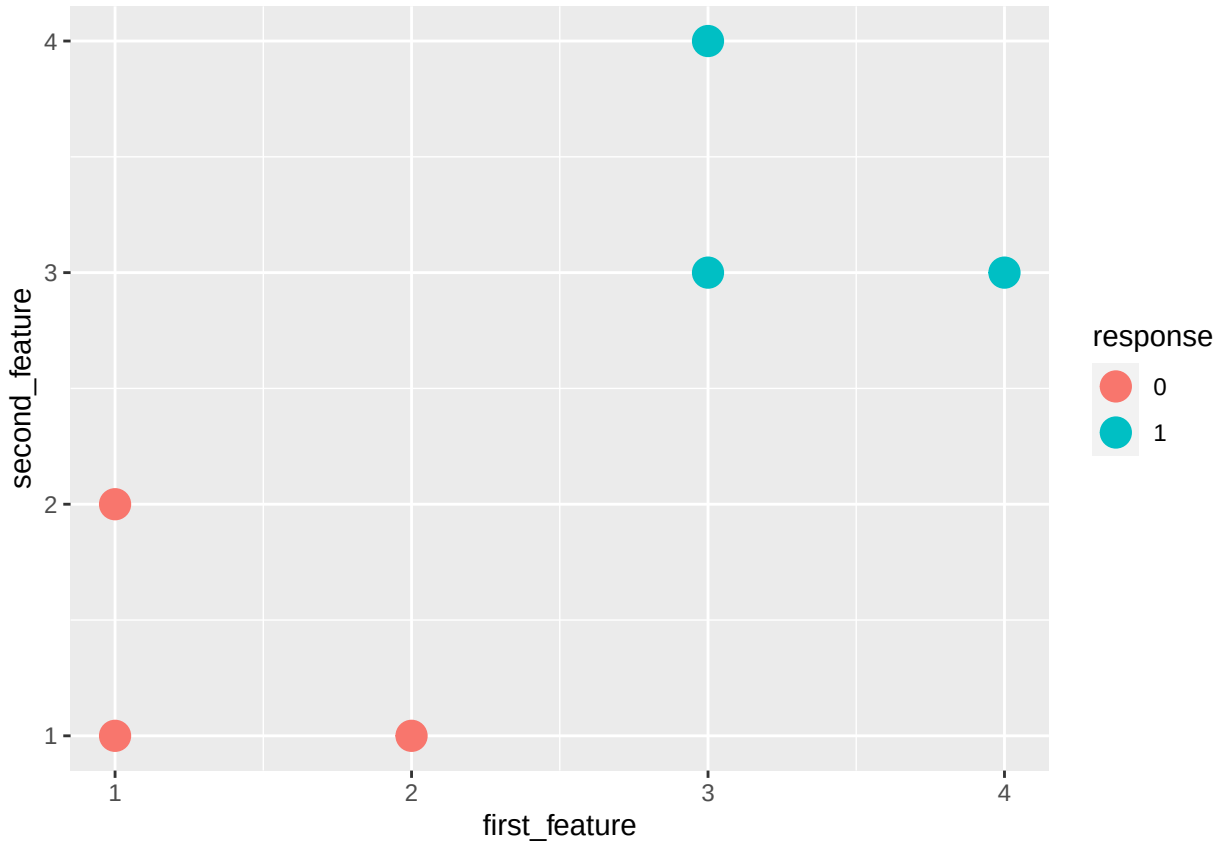
```
      x_part = w[2:length(w)] * Xinput

      d = 0.5 - y_part * (x_part - w[1])

      sum_hinge_errors = sum(max(0, d))
      (1 / n) * sum_hinge_errors + lambda * norm_vec(w[2:length(w)])
  }

  # return weight row vector with p + 1. Includes bias
  optim_df = optimx(w, cost_fn, method="nlm", itnmax=MAX_ITER)

  c(optim_df$p1, optim_df$p2, optim_df$p3)
}
```

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:

```
X_simple_feature_matrix = as.matrix(cbind(Xy_simple$first_feature, Xy_simple$second_feature))
y_binary = as.matrix(as.numeric(Xy_simple$response))


svm_model_weights = linear_svm_learning_algorithm(X_simple_feature_matrix, y_binary)

print(svm_model_weights)
```

```
## [1] -7.235957e+00 -3.986292e-08 -2.769437e-07
```

```
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:

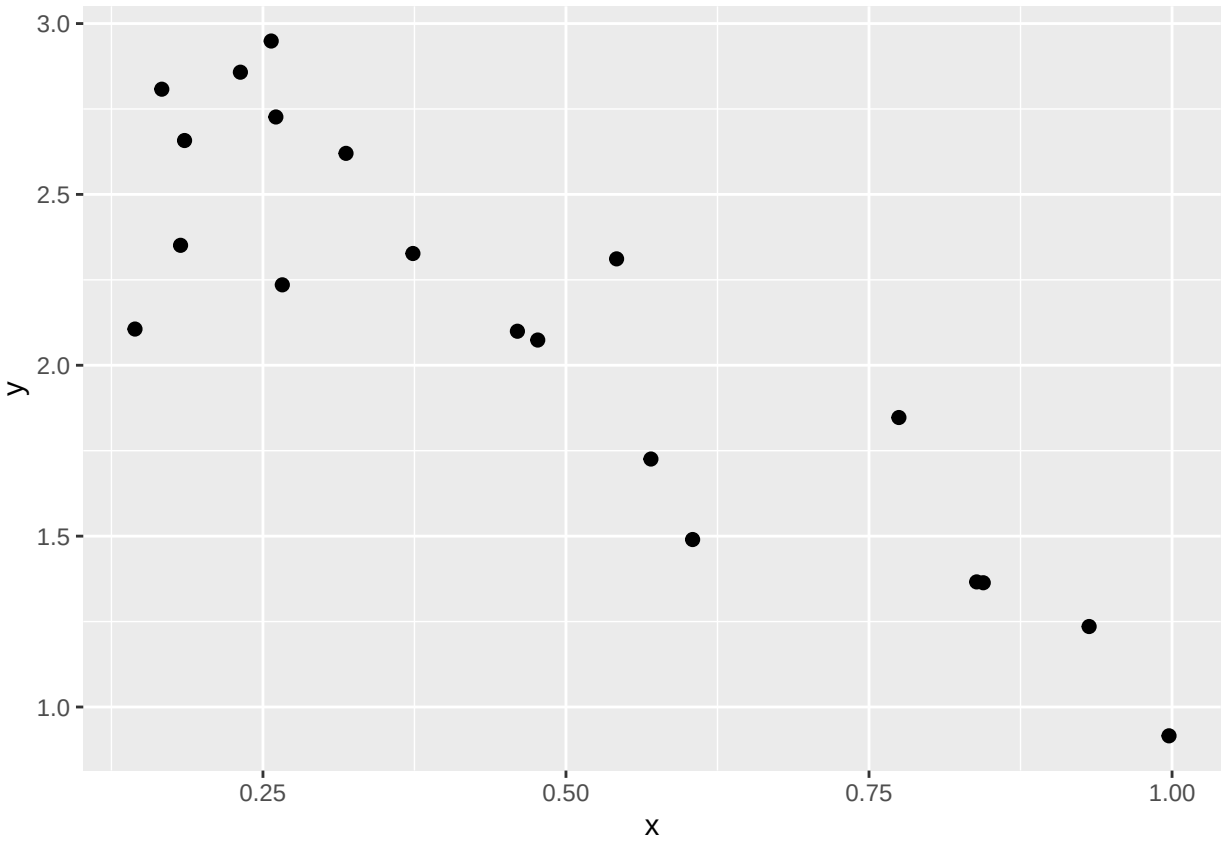
```
n = 20
x = runif(n)
beta_0 = 3
beta_1 = -2
```

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 $beta_1$?

Yes since epsilon is not related to x or y at all. This creates a random align that is not near the plotted data. For that reason, we need to train the alogrithm to get to know the data so it could make a reasonable line.

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 (length(x) != n) {
    stop("x and n must be 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("Dimensions of n must be greater than 2")
```

```
  }


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


  model = list(b_0 = b_0, b_1 = b_1, y_hat = y_hat, e = e, SSE = SSE, SST = SST, MSE =    MSE, RMSE = RM

  class(model) = "my_simple_ols_obj"

  model
}
```

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

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

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

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

```
X = cbind(rep(1, n), x)
```

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 0.5417602
## 2            1 0.1665459
## 3            1 0.5700734
## 4            1 0.1444282
## 5            1 0.7746575
## 6            1 0.2606698
## 7            1 0.1820354
## 8            1 0.9315848
## 9            1 0.2659844
## 10           1 0.8387981
```

```
## 11           1 0.4599279
## 12           1 0.3185263
## 13           1 0.3737227
## 14           1 0.2313942
## 15           1 0.6044668
## 16           1 0.1853325
## 17           1 0.4767080
## 18           1 0.2568025
## 19           1 0.8442367
## 20           1 0.9974960
## attr(,"assign")
## [1] 0 1
```

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

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

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

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

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

```
beta_0 = 3
beta_1 = -2
beta = c(beta_0, beta_1)

ns = 10^(1:6)
errors_in_b = array(NA, length(ns))
for (i in 1 : length(ns)) {
  n = ns[i]
  x = runif(n)
  h_star_x = beta_0 + beta_1 * x
  epsilon = rnorm(n, mean = 0, sd = 0.33)
  y = h_star_x + epsilon

  mod = my_simple_ols(x, y)
  b = c(mod$b_0, mod$b_1)
  errors_in_b[i] = sum((beta - b)^2)
}
log(errors_in_b)
```

```
## [1]  -2.039849  -9.548293  -5.015635  -7.847635 -13.318648 -13.804674
```

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)
```

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
```

There are 928 rows (n = 928) and 2 columns (p = 1, since one is the variable we are predicting). The variable "parent" represent the average height of the mother and father. The variable "child" represent the height of the respective mother and father. You can see that the mean height of the parent column is 68.2 inches.

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.

```
mod = lm(child~parent, Galton)
b_0 = coef(mod)[1]
b_1 = coef(mod)[2]
summary(mod)$sigma
```

```
## [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: a average of parent height 0, will yield a child whose height is 23.94 b_1: represents a rate at which a child height growths as the height of their parents also grows RMSE: This represents the mean height error of all the observations R^2: What percent of the variance could be explained by the model. If the entire variance is explained by the model (100% rsquared) then all the data points fall in the fitted line. In this case the model could only explain 21% of the variance which is okay because the phenomenon we are studying relies on an array of other factors as well.

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

Considering the simplicity of our data, we could say that the model performs okay We know that if we wanted to model the correlation between parents and childs heights we would need more features that can help us fit a better line. This model only considers one feature thus we could say that there is a high error due to ignorance.

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.

Genetically, we know that features such as height are passed from parents down to their children. So assuming that there is some correlation between parents and child height is reasonable. But what this fitted line shows, is that parents height might not be sufficient enough, and we can see this is the R^2.

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?

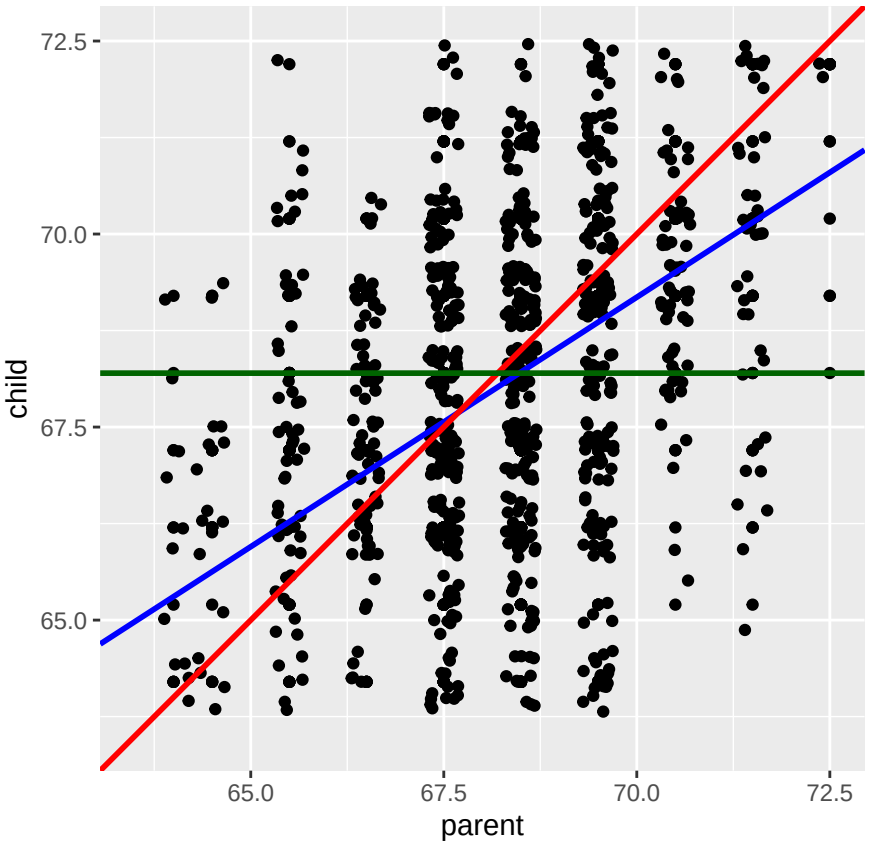
beta_0 would be 0 and beta_1 would be 1 because every x value with be equal to the y value and it would create a linear line with intercept 0 and slope 1.

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 = 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 93 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 smaller 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 toward mediocrity in hereditary stature" because he realized that as he fitted the line, the heights will reverse back to the mean.

Why should this effect be real?

Our best model without any features is to take the mean of our response variable, so it makes sense that our model should be something similar

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.

Because when the model fits the data, the y variable tends to reverse itself back to the mean. You can call it "smoothed linear representations" or "continuous linear representation".

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 = sample(1:5, size = 10, replace = TRUE)
y = sample(1:5, size = 10, replace = TRUE)
Xy = data.frame(x = x, y = y)
mod = lm(Xy$x ~ Xy$y, data=Xy)
summary(mod)$r.squared
```

```
## [1] 0.1487603
```

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

```
## [1] 1.134681
```

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 = sample(1:10, size = 10, replace = TRUE)
y = rep(1, 10)
Xy = data.frame(x = x, y = y)
mod = lm(Xy$x ~ Xy$y, data=Xy)
summary(mod)$r.squared
```

```
## [1] 0
```

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

```
## [1] 3.084009
```

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 = nrow(X) # get the num of rows in X
  # get the num of cols in X. Add 1 since we add the intercept later
  p = ncol(X) + 1
  # check that X and y have the same number of rows
  if (nrow(X) != nrow(y)) {
    stop("X and y must have the same number of rows")
  }
  # check that X is numeric and integer
  if (class(X)  != "numeric" & class(X) != "integer") {
    stop("x needs to be numeric")
  }
  # check that y is numeric and integer
  if (class(y)  != "numeric" & class(y) != "integer") {
    stop("y needs to be numeric")
  }
  # check that there are more than 2 rows
  if (n <= 2) {
    stop("Dimensions of n must be greater than 2")
  }
  # add the intercept columns so now we have p + 1 columns
  X = as.matrix(cbind(1, X[, 1:p]))
  colnames(X)[1] = "(intercept)"
  # take the dot product of the transpose and the orig matrix
```

```
  XtX = t(X) %*% X
  # check that this is invertable first
  # and check that is full rank
  tryCatch({
    XtXinv = solve(XtX)
  }, error = function(e) {
    stop("Matrix is not invertible or does not have full rank")
  })

  b = XtXinv %*% t(X) %*% y
  yhat = X %*% b # predictions of the training data
  e = y - yhat # residuals
  df  = ... # degrees of freedom
  SSE = t(e) %*% e
  s_sq_y = var(y )
  SST = (n -1)  * s_sq_y
  MSE = 1 /(nrow(X) - ncol(X)) * SSE
  RMSE = sqrt(MSE)
  Rsq = 1 - SSE / SST

  model = list("b"=b, "yhat"=yhat, "e"=e, "SSE"=SSE, "SST"=SST, "MSE"=MSE, "RMSE"=RMSE, "Rsq"=Rsq)

  class(model) = "my_ols"

  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
anova_mod = lm(Price ~ Type, cars)
print(coef(anova_mod))
```

```
## (Intercept)   TypeLarge TypeMidsize   TypeSmall  TypeSporty     TypeVan
##   18.212500    6.087500    9.005682   -8.045833    1.180357    0.887500
```

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)

  x_start %*% model$b
}
```