Lab 3

Mohammed Hasan

11:59PM February 15

Perceptron

You will code the "perceptron learning algorithm" for arbitrary number of features p. Take a look at the comments above the function. Respect the spec below:

```r
#' # Title
#' Perception Learning Algorithm
#'
#' # DESCRIPTION
#' Returns a vector that represents a hyper plane that linearly separates
#' the binary response in the input space
#'
#' @param Xinput      n * p matrix of characteristics / features of the training data
#' @param y_binary    vector of length n of the binary responses
#' @param MAX_ITER    Maximum iterations of the algorithm
#' @param w_0           p + 1 length vector represents initial state
#'
#' @return            The computed final parameter (weight) as a vector of length p + 1
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w_0 = NULL){
  p = ncol(Xinput)
  n = nrow(Xinput)

  # We are cbinding because the first w_0 is the intercept
  Xinput = cbind(1, Xinput)

  # If user wants to use default we make w_0 a vector of 0
  if(is.null(w_0)){
    w_0 = rep(0, p+1)
  }

  w_prev = w_0
  w = w_0

  for(iter in 1 : MAX_ITER){ # Step 4 : Repeat n amount of times
    for(i in 1 : n){ # Step 3 : Repeat step 1, 2
      # Step 1: Getting a prediction for each subject
      y_hat_i = ifelse(sum(w_prev * Xinput[i,]) >= 0, 1, 0)

      # Step 2: Update weight for each point
      w = w + (y_binary[i] - y_hat_i) * Xinput[i,]

    }#end-inner-j
    if(identical(w, w_prev)){
      break # TE = 0
    }#end-if

    w_prev = w
  }#end-outer-i

  w

}
```

To understand what the algorithm is doing - linear “discrimination” between two response categories, we can draw a picture. First let’s make up some very simple training data D.

```
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
)
```

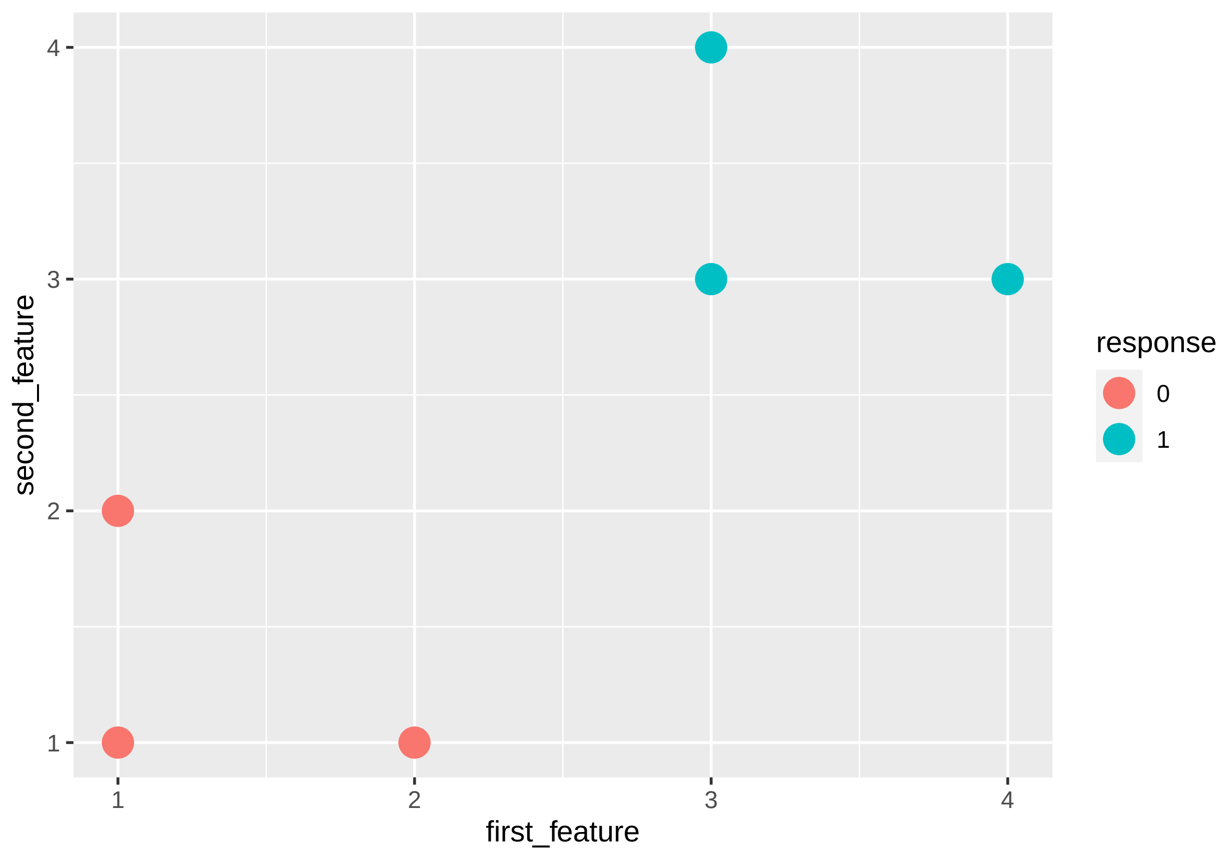
We haven’t spoken about visualization yet, but it is important we do some of it now. Thus, I will write this code for you and you will just run it. First we load the visualization library we’re going to use:

```
pacman::p_load(ggplot2)
```

We are going to just get some plots and not talk about the code to generate them as we will have a whole unit on visualization using `ggplot2` in the future.

Let’s first plot y by the two features so the coordinate plane will be the two features and we use different colors to represent the third dimension, y.

```
simple_viz_obj = ggplot(Xy_simple, aes(x = first_feature, y = second_feature, color = response)) +
  geom_point(size = 5)
simple_viz_obj
```



TO-DO: Explain this picture.

Now, let us run the algorithm and see what happens:

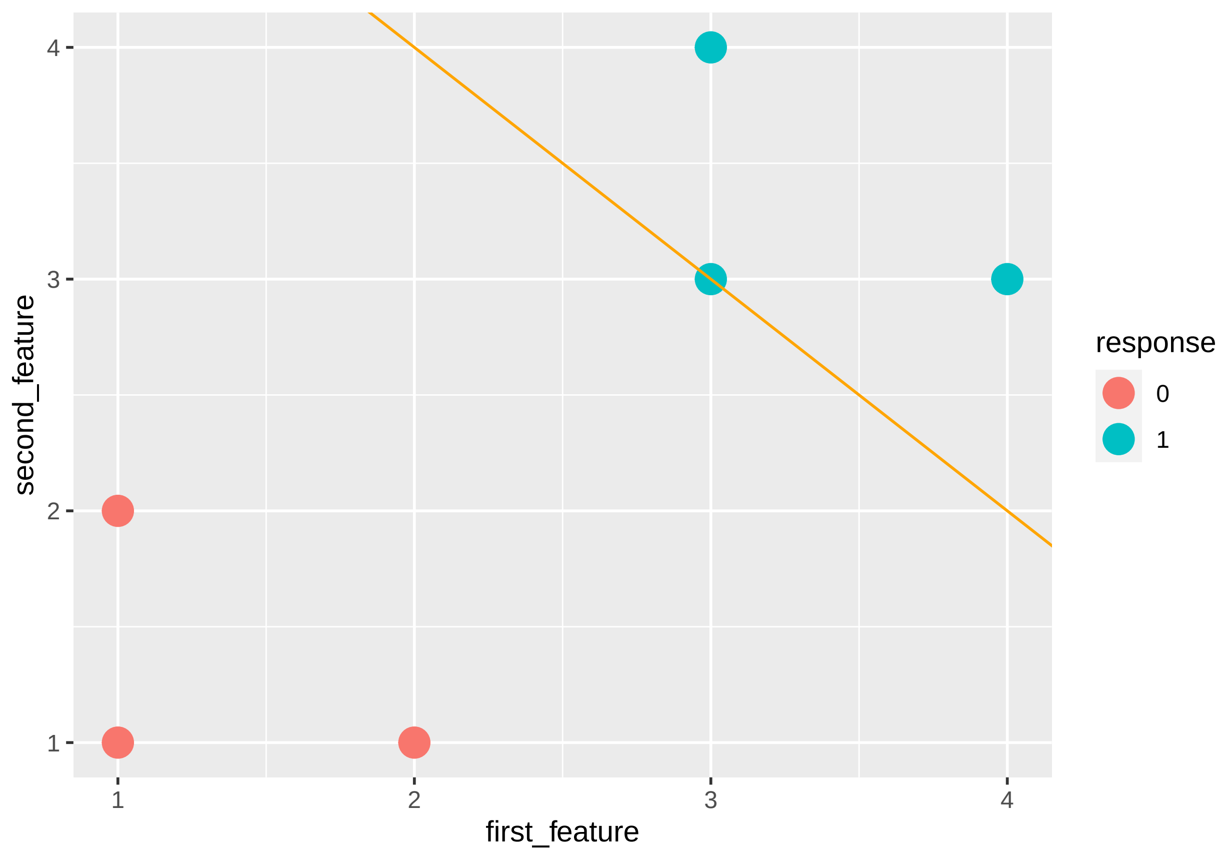
```
w_vec_simple_per = perceptron_learning_algorithm(
  cbind(Xy_simple$first_feature, Xy_simple$second_feature),
  as.numeric(Xy_simple$response == 1))
w_vec_simple_per
```

```
## [1] -12   2   2
```

Explain this output. What do the numbers mean? What is the intercept of this line and the slope? You will have to do some algebra.

TO-DO

```
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
```



Explain this picture. Why is this line of separation not "satisfying" to you?

This line is not optimal because it does not cut the wedge down the middle, which is the max-margin hyperplane.

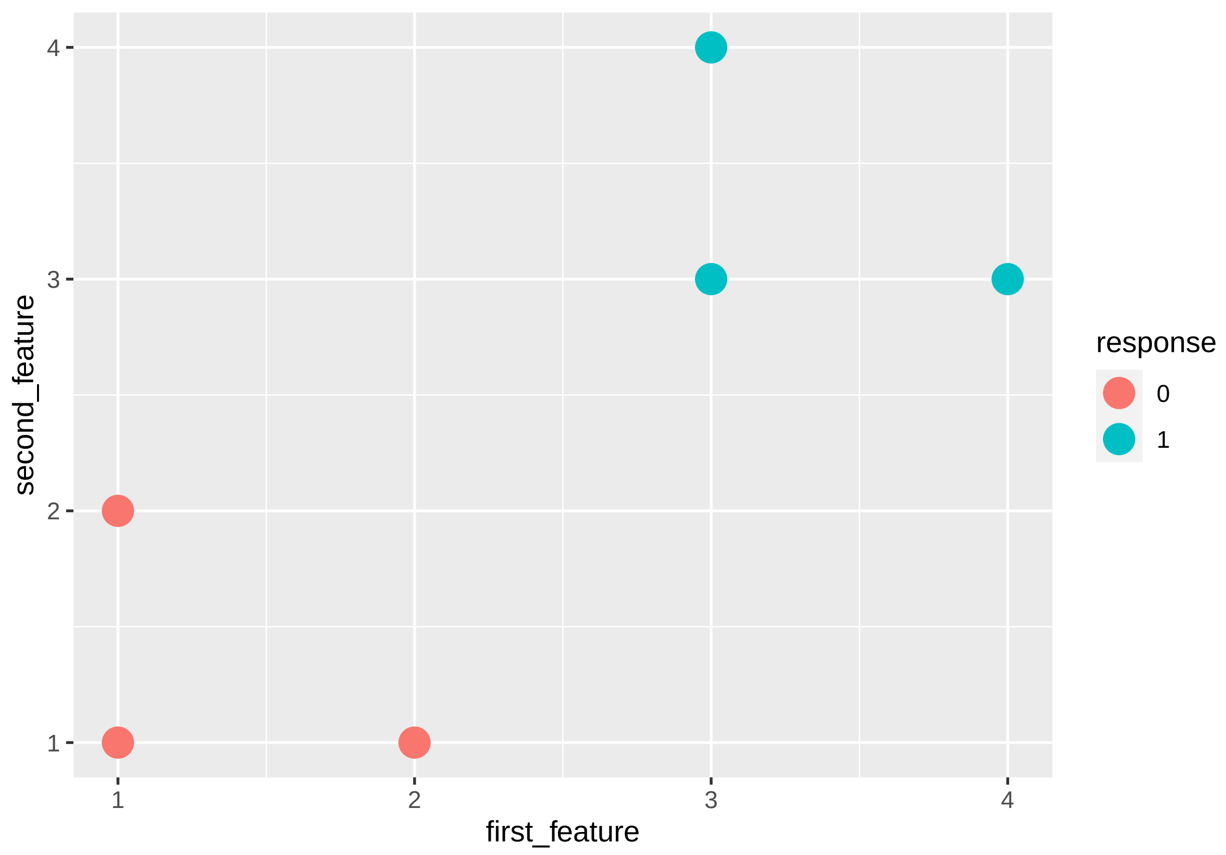
For extra credit, program the maximum-margin hyperplane perceptron that provides the best linear discrimination model for linearly separable data. Make sure you provide ROxygen documentation for this function.

```
#TO-DO
```

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
```

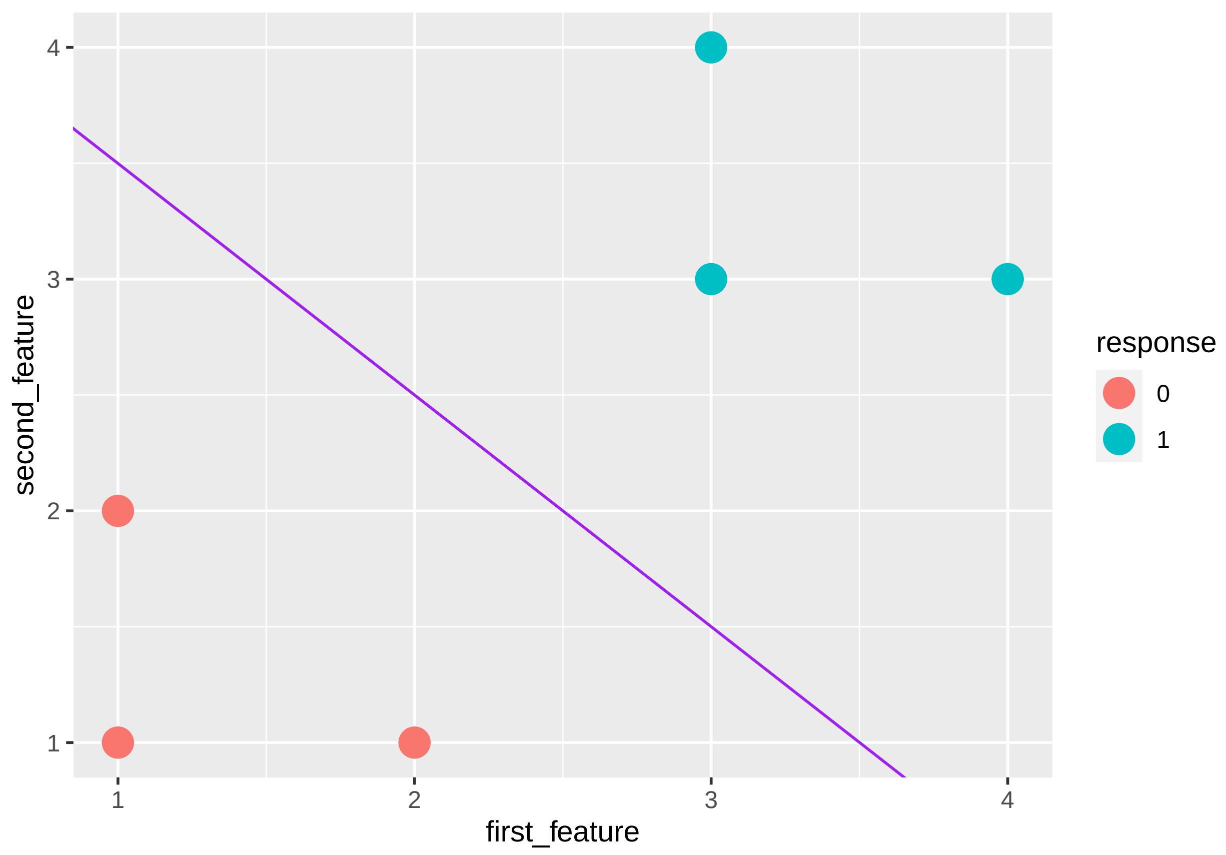


Use the `e1071` package to fit an SVM model to the simple data. Use a formula to create the model, pass in the data frame, set kernel to be `linear` for the linear SVM and don't scale the covariates. Call the model object `svm_model`. Otherwise the remaining code won't work.

```
pacman::p_load(e1071)
svm_model = svm(
  formula = response ~ .,
  data = Xy_simple,
  kernel = "linear",
  scale = FALSE
)
```

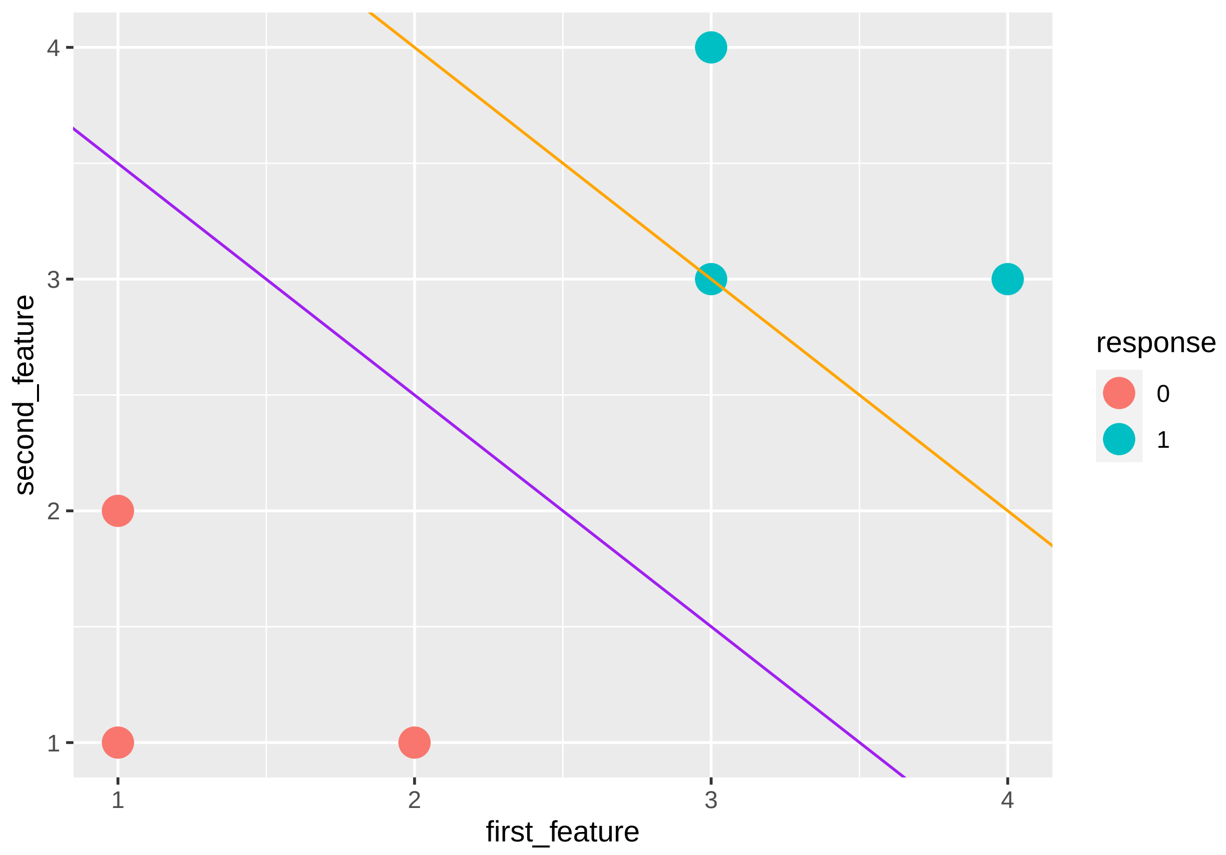
and then use the following code to visualize the line in purple:

```
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:

```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?

Yes, It looks like it splits the wedge in the middle.

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
#'
#' @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){
  #SHE = sum(max (0, half -(y- half)(w dot x_i -b))
  # optimize(w_0, b_0, 1/n SHE + lambda * norm(w)^2, MAX_ITTER) )
}
```

If you are enrolled in 342W the following is extra credit but if you're enrolled in a masters section, 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 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?

TO-DO

Multinomial Classification using KNN

Write a k = 1 nearest neighbor algorithm using the Euclidean distance function. The following comments are standard "Roxygen" format for documentation. Hopefully, we will get to packages at some point and we will go over this again. It is your job also to fill in this documentation.

```
#' Nearest Neighbor Model
#'
#' Classifies its predictions using its nearest neighbors in input space.
#'
#' @param Xinput      Historical measurements
#' @param y_binary    Historical responses
#' @param Xtest       A matrix of measurements where each row is a unit you wish to predict the respons
#' @return            A vector of predicted responses
nn_algorithm_predict = function(Xinput, y_binary, Xtest){
  y_hat = array(NA, nrow(Xtest))
  for (i_star in 1:nrow(Xtest)) {
    # y_binary[which.min((Xinput - Xtest[i,])^2)]
    dsq = array(NA, nrow(Xinput))
    for (i in 1:nrow(Xinput)) {
      dsq[i] = sum((Xinput[i,]-Xtest[istar,])^2)
    }
    y_hat[i_star] = y_binary[which.min(dsq)]
  }
  y_hat
}
```

Write a few tests to ensure it actually works:

```
#TO-DO do at home
```

We now add an argument `d` representing any legal distance function to the `nn_algorithm_predict` function.

Update the implementation so it performs NN using that distance function. Set the default function to be the Euclidean distance in the original function. Also, alter the documentation in the appropriate places.

```
#' Nearest Neighbor Model
#'
#' Classifies its predictions using its nearest neighbors in input space.
#'
#' @param Xinput      Historical measurements
#' @param y_binary    Historical responses
#' @param Xtest       A matrix of measurements where each row is a unit you wish to predict the respons
#' @param d           distance function to measure distances between two observations which defaults to
#' @return            A vector of predicted responses
nn_algorithm_predict = function(Xinput, y_binary, Xtest, d = function(v_1, v_2){
  sum((v_1 - v_2)^2)
}){
  y_hat = array(NA, nrow(Xtest))
  for (i_star in 1:nrow(Xtest)) {
    # y_binary[which.min((Xinput - Xtest[i,])^2)]
    dsq = array(NA, nrow(Xinput))
    for (i in 1:nrow(Xinput)) {
      dsq[i] = d(Xinput[i,],Xtest[i,])
    }
    y_hat[i_star] = y_binary[which.min(dsq)]
  }
  y_hat
}
```

For extra credit (unless you're a masters student), add an argument `k` to the `nn_algorithm_predict` function and update the implementation so it performs KNN. In the case of a tie, choose yhat randomly. Set the default `k` to be the square root of the size of D which is an empirical rule-of-thumb popularized by the "Pattern Classification" book by Duda, Hart and Stork (2007). Also, alter the documentation in the appropriate places.

```
#TO-DO for the 650 students but extra credit for undergrads
```

Regression via OLS with one feature

Let's quickly recreate the sample data set from practice lecture 7:

```
set.seed(1984)
n = 20
x = runif(n)
beta_0 = 3
beta_1 = -2
x
```

```
##  [1] 0.65880473 0.43697503 0.37333816 0.33095629 0.73756366 0.86261016
##  [7] 0.03243676 0.44774443 0.82986892 0.21457412 0.88267976 0.01197508
## [13] 0.70624726 0.71977362 0.20249980 0.02271680 0.29937189 0.66462912
## [19] 0.92160973 0.20576302
```

Compute h^* as `h_star_x`, then draw epsilon from an iid N(0, 0.33^2) distribution as `epsilon`, then compute the vector y.

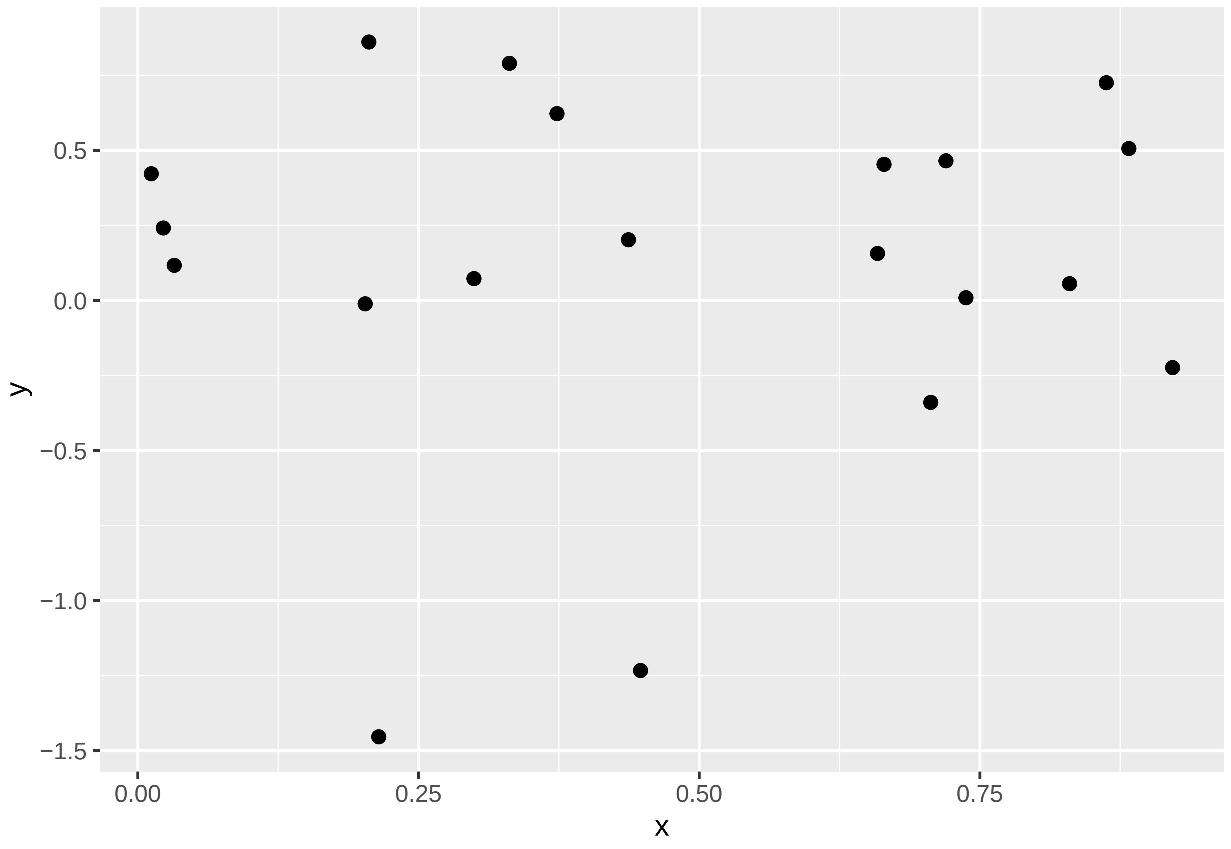
```
#best element beta0 + beta1*x
h_star_x = beta_0 + beta_1 * x
epsilon = rnorm(n, mean = 0, sd = .33)
```

```
y = h_star_x * epsilon
y
```

```
##  [1]  0.156569372  0.202013283  0.622426410  0.790035934  0.008929474
##  [6]  0.725245141  0.116949515 -1.233106658  0.055792466 -1.453919765
## [11]  0.506012048  0.422046731 -0.339794845  0.465202196 -0.011319356
## [16]  0.241331487  0.072678797  0.453362349 -0.223816028  0.861020143
```

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 the intercept and slope match

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){
  ols_obj = list()
```

```r
  if (class(x) != "numeric") {
    stop("x is not numeric")
  }
  if (class(y) != "numeric") {
    stop("y is not numeric")
  }

  n = length(x)

  if (length(y != n)) {
    stop("x and y are not the same length")
  }
  x_bar = mean(x)
  y_bar = mean(y)
  b_1 = sum((x - x_bar) * (y - y_bar)) / sum((x - x_bar)^2) #sum of x_i and y_i
  b_0 = y_bar - b_1 * x_bar
  e = y - (b_0 + b_1*x)


  SSE = sum(e^2)
  MSE = SSE / (n-1)
  RMSE = sqrt(MSE)
  SST = sum((y - y_bar)^2)
  RSQ = (SST - SSE) / SST

  class(ols_obj) = "my_simple_ols_obj"
  ols_obj

  ols_obj$b_1 = b_1
  ols_obj$b_0 = b_0
  ols_obj$e = e
  ols_obj$SSE = SSE
  ols_obj$MSE = MSE
  ols_obj$RMSE = RMSE
  ols_obj$SST = SST
  ols_obj$RSQ = RSQ

  ols_obj
}
```

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$RMSE$ 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(0, (my_simple_ols_mod$e), tol = 1e-4)

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

```r
X = cbind(1, x)
X
```

```
##                    x
##  [1,] 1 0.65880473
##  [2,] 1 0.43697503
##  [3,] 1 0.37333816
##  [4,] 1 0.33095629
##  [5,] 1 0.73756366
##  [6,] 1 0.86261016
##  [7,] 1 0.03243676
##  [8,] 1 0.44774443
##  [9,] 1 0.82986892
## [10,] 1 0.21457412
## [11,] 1 0.88267976
## [12,] 1 0.01197508
## [13,] 1 0.70624726
## [14,] 1 0.71977362
## [15,] 1 0.20249980
## [16,] 1 0.02271680
## [17,] 1 0.29937189
## [18,] 1 0.66462912
## [19,] 1 0.92160973
## [20,] 1 0.20576302
```

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

```
X_model_matrix = model.matrix(~ x)
X_model_matrix
```

```
##    (Intercept)          x
## 1            1 0.65880473
## 2            1 0.43697503
## 3            1 0.37333816
## 4            1 0.33095629
## 5            1 0.73756366
## 6            1 0.86261016
## 7            1 0.03243676
## 8            1 0.44774443
## 9            1 0.82986892
## 10           1 0.21457412
## 11           1 0.88267976
## 12           1 0.01197508
## 13           1 0.70624726
## 14           1 0.71977362
## 15           1 0.20249980
## 16           1 0.02271680
## 17           1 0.29937189
## 18           1 0.66462912
## 19           1 0.92160973
## 20           1 0.20576302
## 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_mod, x_star){
  b_0 = my_simple_ols_mod$b_0
  b_1 = my_simple_ols_mod$b_1
  y_pred <- b_0 + b_1 * x_star

  return(y_pred)
}
```

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 beta_0 and beta_1. How about ||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 = seq(10, 1000, by=10)
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

  lm_fit = lm(y ~ x)
  b = coef(lm_fit)

  errors[i] = sum((beta - b)^2)
}
errors
```

```
##   [1] 4.466601e-01 1.692199e-02 3.764019e-02 4.044529e-02 6.796481e-03
##   [6] 1.298653e-02 1.532715e-02 4.128727e-02 3.166151e-02 2.837126e-02
##  [11] 7.870717e-03 2.907036e-03 6.348268e-04 1.836077e-03 8.456102e-03
##  [16] 1.121750e-02 2.455344e-02 7.656971e-03 1.507810e-02 1.457101e-02
##  [21] 5.245934e-03 7.679942e-03 2.362615e-03 1.911212e-03 3.899928e-03
##  [26] 2.606265e-04 4.323928e-04 4.438675e-03 3.720698e-03 8.788867e-03
##  [31] 2.565628e-03 3.108651e-03 2.090733e-03 2.994216e-03 5.385226e-03
##  [36] 3.295846e-03 9.829141e-04 6.380727e-03 6.700906e-04 1.104411e-03
##  [41] 7.767145e-04 3.429222e-03 7.858291e-05 5.296502e-04 6.324495e-04
##  [46] 1.343749e-02 1.277292e-03 8.282289e-04 4.764606e-04 5.170953e-04
##  [51] 8.603062e-04 2.082451e-03 9.867292e-04 1.590643e-02 1.614136e-02
##  [56] 1.620104e-03 8.127011e-03 1.183758e-03 2.106212e-04 1.150262e-02
##  [61] 3.575879e-03 1.893558e-03 8.309170e-04 1.822879e-03 4.989799e-04
##  [66] 1.970373e-03 7.259231e-04 5.081066e-03 5.318643e-03 1.237833e-04
##  [71] 1.901324e-03 1.005183e-03 3.469159e-03 1.164904e-04 6.434560e-04
##  [76] 4.152169e-04 9.031056e-04 3.381119e-03 7.155959e-05 4.060478e-05
##  [81] 1.441954e-03 1.990030e-04 7.246262e-04 1.209876e-03 1.428216e-04
##  [86] 3.304759e-03 5.938914e-06 1.812258e-03 1.596784e-03 1.740893e-04
##  [91] 1.610623e-03 1.198305e-03 8.458444e-03 5.426149e-05 3.587186e-03
##  [96] 6.426378e-04 4.975765e-04 2.753174e-03 6.955335e-05 1.218971e-04
```

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)

```r
pacman::p_load(HistData)
```

In it, there is a dataset called `Galton`. Load it up.

```r
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

```r
pacman::p_load(skimr)
summary(Galton)
```

```
##      parent          child      
##  Min.   :64.00   Min.   :61.70  
##  1st Qu.:67.50   1st Qu.:66.20  
##  Median :68.50   Median :68.20  
##  Mean   :68.31   Mean   :68.09  
##  3rd Qu.:69.50   3rd Qu.:70.20  
##  Max.   :73.00   Max.   :73.70
```

The mean height for parents is reported as 68.31 inches, while the mean height for children is slightly lower at 68.09 inches.The maximum height recorded for parents is 73.00 inches, whereas for children, it is slightly higher at 73.70 inches. This observation suggests that while there is a tendency for heights to regress towards the mean.

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

```r
avg_height = 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?

```r
sd(Galton$child)
```

```
## [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.

```r
lm_mod = lm(child ~ parent, data = Galton)
coefs = coefficients(lm_mod)
b_0 = coefs[1]
b_1 = coefs[2]
summary(lm_mod)$sigma
```

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

```r
summary(lm_mod)$r.square
```

```
## [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.

it is a reasonable assumption. Meaning the slope should be 1.

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

Rˆ2 of 0.21, the model explains only a portion of the variance in children's heights based on parents' heights. RMSE of 2.24 inches further indicates there's a notable average prediction error. This means that it probably isnt a good model.

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.

Since children inherit genes from their parents, it biologically makes sense that they would have the same height.

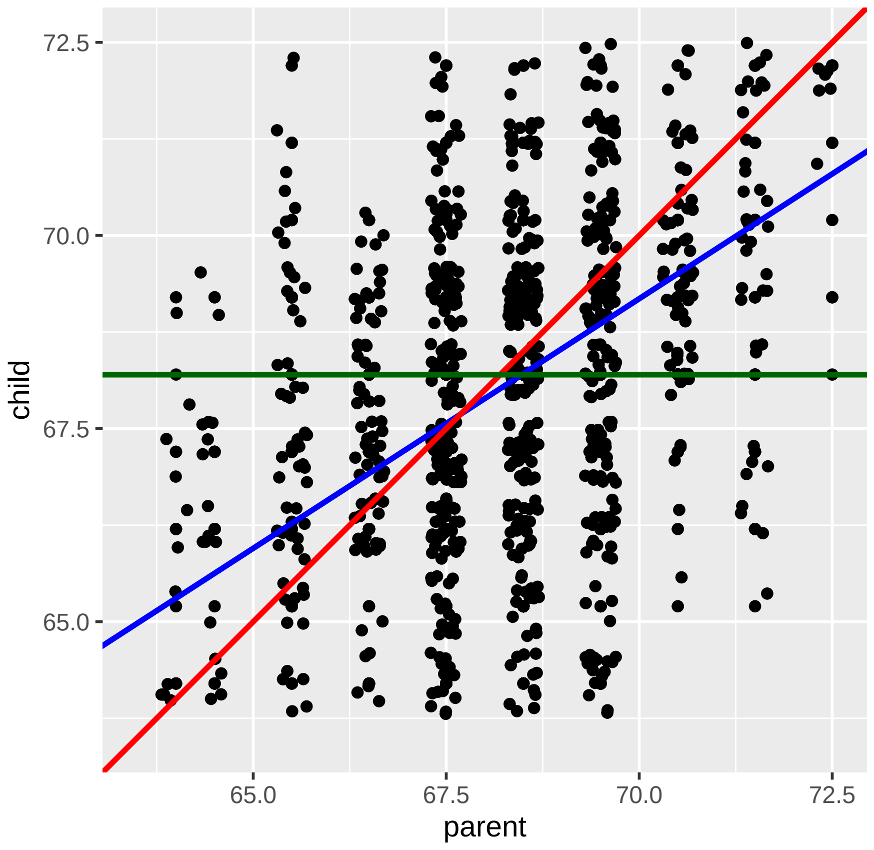
If they were to have the same height and any differences were just random noise with expectation 0, what would the values of beta_0 and beta_1 be?

beta_1 should ideally be 1.This is because it should be a 1-to-1 relationship linearly. beta_0 would be 0, indicating no fixed amount to add or subtract from the parent's height to predict the child's height.

Let's plot (a) the data in D as black dots, (b) your least squares line defined by b_0 and b_1 in blue, (c) the theoretical line beta_0 and beta_1 if the parent-child height equality held in red and (d) the mean height in green.

```r
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: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.

## Warning: Removed 76 rows containing missing values (`geom_point()`).

## Warning: Removed 86 rows containing missing values (`geom_point()`).
```



Fill in the following sentence:

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

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

because shorter parents had their children grow taller towards the mean, meanwhile tall parents had their children grow shorter towards the mean.

Why should this effect be real?

Biologically, this should be real because recreating humans is not creating clones, instead you are creating a human who inherits many traits combining into an average human.

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.

Regression is used to model a continous y given known x's, dependent variable y is continuous, distinguishing this class of models from those designed for categorical outcomes or classification problems.

You can now clear the workspace.

```r
rm(list = ls())
```

Create a dataset D which we call `Xy` such that the linear model has R^2 about 50% and RMSE approximately 1.

```r
n= 50
x = runif(n)
y = 0 + 2*x + rnorm(n, mean = 0, sd = 1)
Xy = data.frame(x = x, y = y)
```

Create a dataset D which we call `Xy` such that the linear model has R^2 about 0% but x, y are clearly associated. YOU DONT HAVE TO DO THIS ONE

```r
x = seq(from = 0, to = 100, length.out = 100)
y = sin(x) + rnorm(100, mean = 0, sd = 0.2)
Xy = data.frame(x = x, y = y)
```

Extra credit but required for 650 students: create a dataset 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.

```r
epsilon = 0.01
M = 1000
#TO-DO
```