Lab 3

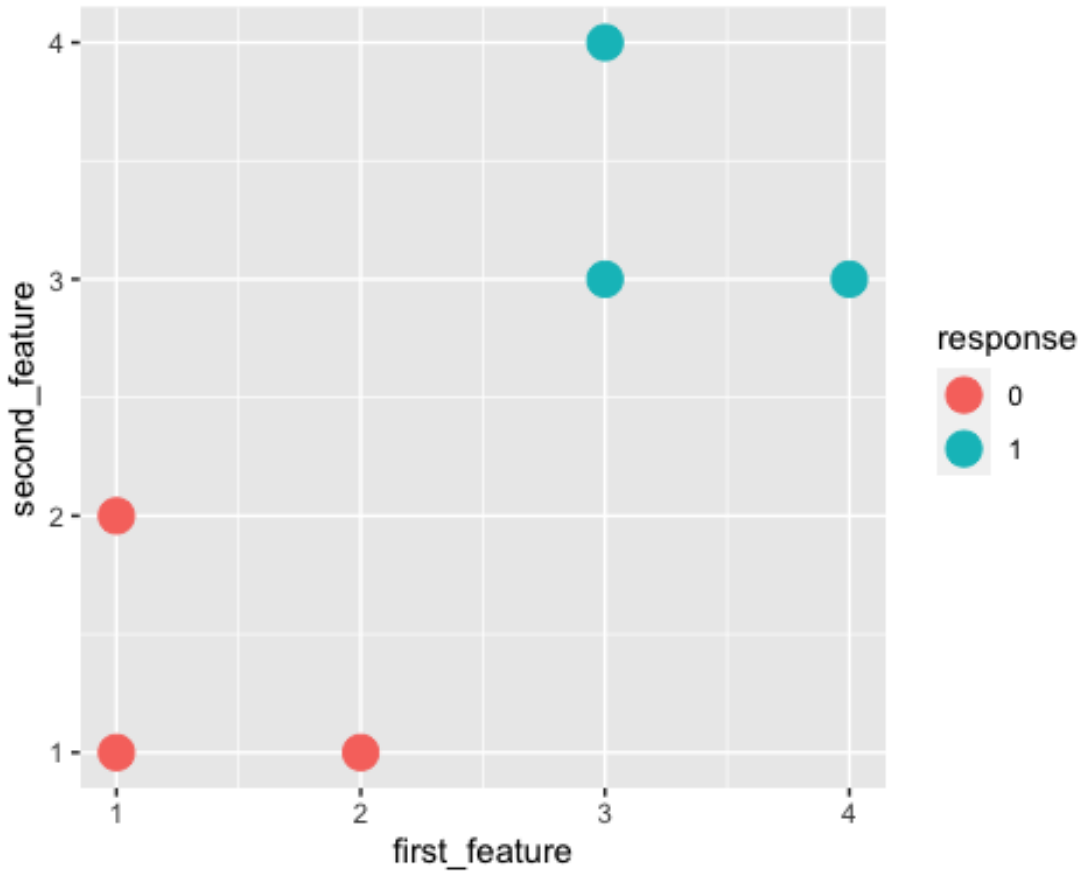
Nasif Khan

11:59PM March 4, 2021

Support Vector Machine vs. Perceptron

We recreate the data from the previous lab and visualize it:

```
x <- 0
y <- 0
color <- 0
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
```



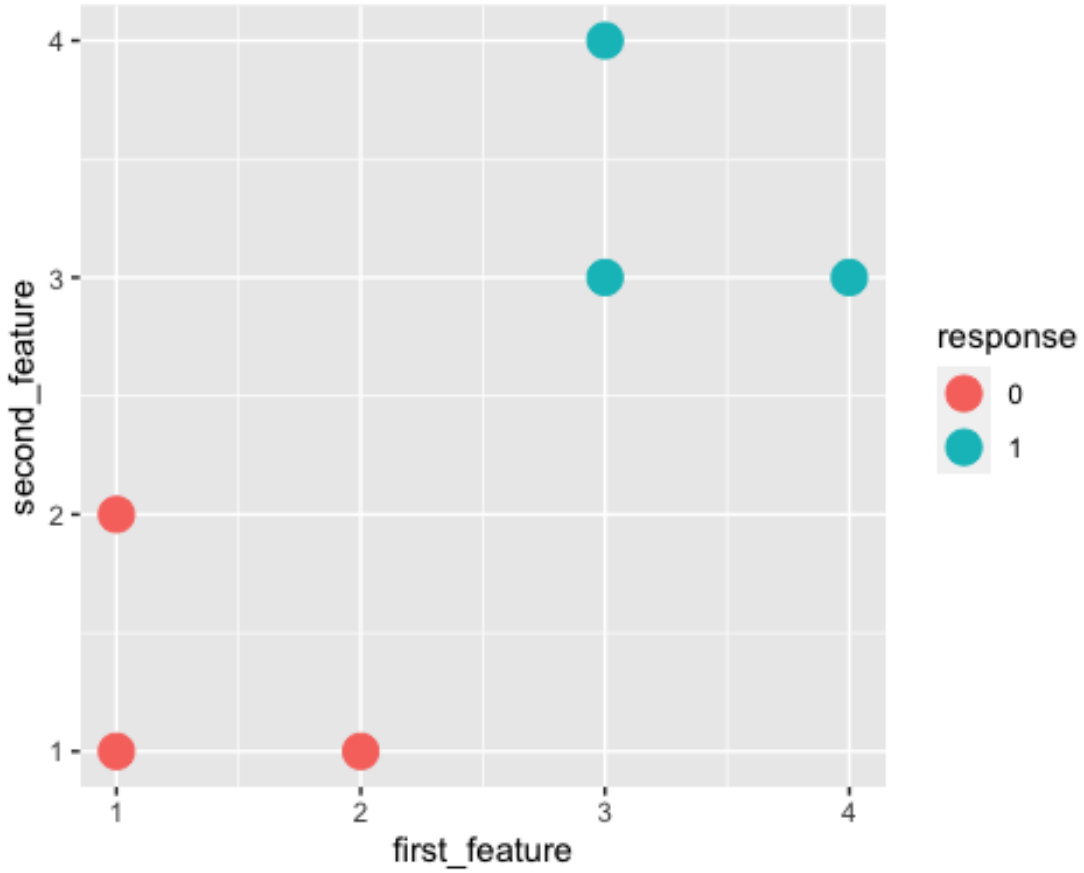
Use the `e1071` package to fit an SVM model to the simple data. Use a formula to create the model, pass in the data frame, set kernel to be `linear` for the linear SVM and don't scale the covariates. Call the model object `svm_model`. Otherwise the remaining code won't work.

```
pacman::p_load(e1071)
svm_model = svm(
  formula = second_feature ~ .,
  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],
```

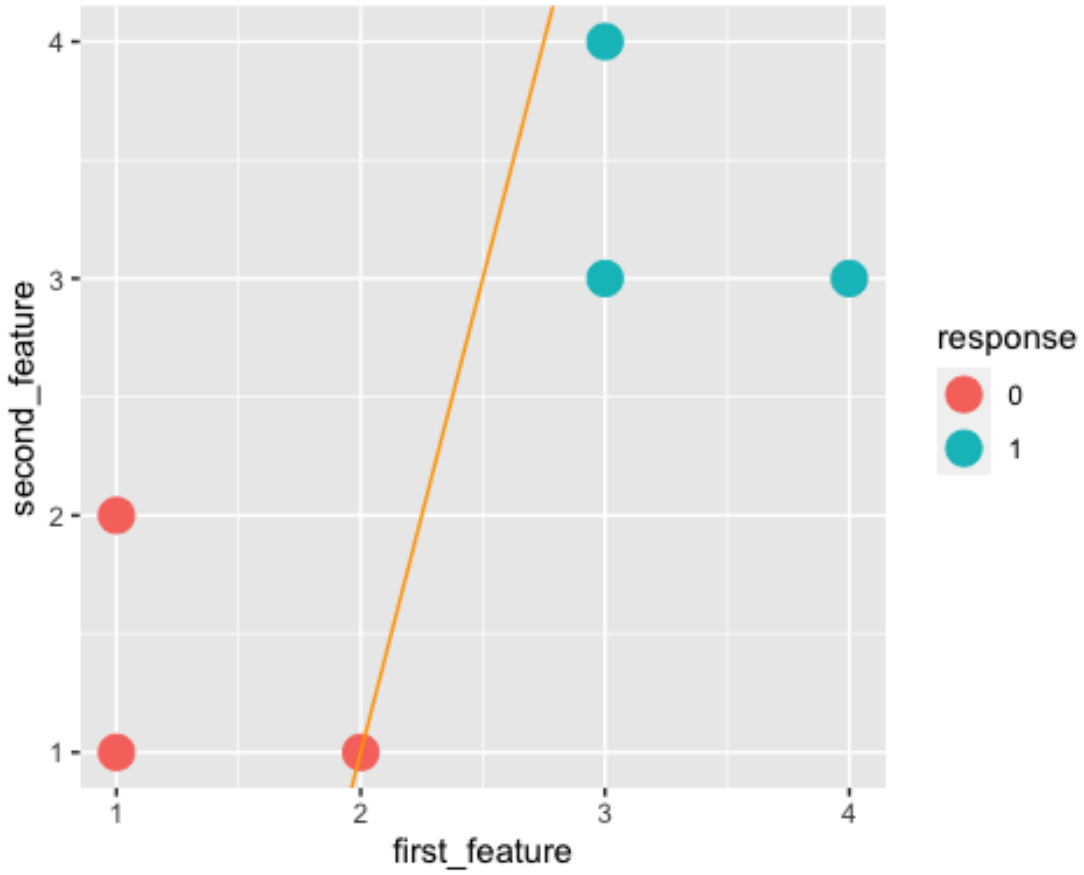
```
    color = "purple")
simple_viz_obj + simple_svm_line
```



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(first_feature, second_feature,
MAX_ITER = 1000, w = NULL){

  X1 = cbind(1, first_feature)
    if(is.null(w)){
           w = rep(0, ncol(X1))
    }
for (iter in 1 : MAX_ITER){
 for (i in 1 : nrow(first_feature)){
       Xi = X1[i, ]
       y_hat = ifelse( Xi %*% w > 0,1, 0)
       w = w + as.numeric(second_feature[i] - y_hat) * Xi

     }
    }

    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? No.

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){
  #for i in range:nrow(i){
    #
#  }
#}
```

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).

```
{r} # #' This function implements the hinge-loss + maximum margin linear
support vector machine algorithm of Vladimir Vapnik (1963). # #' # #' @param
Xinput      The training data features as an n x p matrix. # #' @param
y_binary    The training data responses as a vector of length n consisting of
only 0's and 1's. # #' @param MAX_ITER    The maximum number of iterations
the algorithm performs. Defaults to 5000. # #' @param lambda      A scalar
hyperparameter trading off margin of the hyperplane versus average hinge
loss. # #'                    The default value is 1. # #' @return
The computed final parameter (weight) as a vector of length p + 1 #
linear_svm_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 5000,
lambda = 0.1){ #   #TO-DO # } #
```

If you wrote code (the extra credit), run your function using the defaults and plot it in brown vis-a-vis the previous model's line:

```
{r} # 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

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:

```
{r} # 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 $\y$.

```
{r} # h_star_x = #TO-DO  # epsilon = #TO-DO # y = #TO-DO #
```

Graph the data by running the following chunk:

```
{r} # pacman::p_load(ggplot2) # simple_df = data.frame(x = x, y = y) #
simple_viz_obj = ggplot(simple_df, aes(x, y)) +  #   geom_point(size = 2) #
simple_viz_obj #
```

Does this make sense given the values of $beta_0$ and $beta_1$?

Write a function `my_simple_ols` that takes in a vector `x` and vector `y` and returns a list that contains the `b_0` (intercept), `b_1` (slope), `yhat` (the predictions), `e` (the residuals), `SSE`, `SST`, `MSE`, `RMSE` and `Rsq` (for the R-squared metric). Internally, you can only use the functions `sum` and `length` and other basic arithmetic operations. You should throw errors if the inputs are non-numeric or not the same length. You should also name the class of the return value `my_simple_ols_obj` by using the `class` function as a setter. No need to create ROxygen documentation here.

```{r} # my_simple_ols = function(x, y){ #   #TO-DO # } #
```

```{r} # g = function(x_star, X, y){ #   #TO-DO # } #
```