# Lab 3

*Connor Brown*

*11:59PM February 24, 2019*

## Perceptron

You will code the "perceptron learning algorithm". Take a look at the comments above the function. This is 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.

```
#' perceptron_learning_algorithm
#'
#' The perceptron learning algorithm creates a line between two linearly separable sets of data (classi
#'
#' @param Xinput      An nxp matrix where p is # of features and n is # of observations
#' @param y_binary    A binary vector of length n which contains the output of your testing data
#' @param MAX_ITER    The number of iterations you want; defaults to 1000.
#' @param w           Initial weight vector (numeric) of length p+1.
#'
#' @return            The computed final parameter (weight) as a vector of length p + 1
#' @export            [In a package, this documentation parameter signifies this function becomes a pub
#'
#' @author            [Connor Brown]
perceptron_learning_algorithm = function(X_input, y_binary, MAX_ITER = 1000, w = NULL){

  n = nrow(X_input)
  p = ncol(X_input)

  if(is.null(w)){
    w = rep(0, p+1) #zero vector of length p+1
  }

  X_input = cbind(1, X_input) #Adds column of 1's to X_input matrix

  for (iter in 1 : MAX_ITER){
    for (i in 1 : n){
      x_i = X_input[i, ]
      yhat_i = ifelse(sum(x_i * w) >= 0, 1, 0)
      y_i = y_binary[i]
      e_i = y_i - yhat_i


      for(j in 1 : (p+1)){
        w[j] = w[j] + (e_i * x_i[j])
      }
    }
  }
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 $\mathbb{D}$.

```
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
)
```

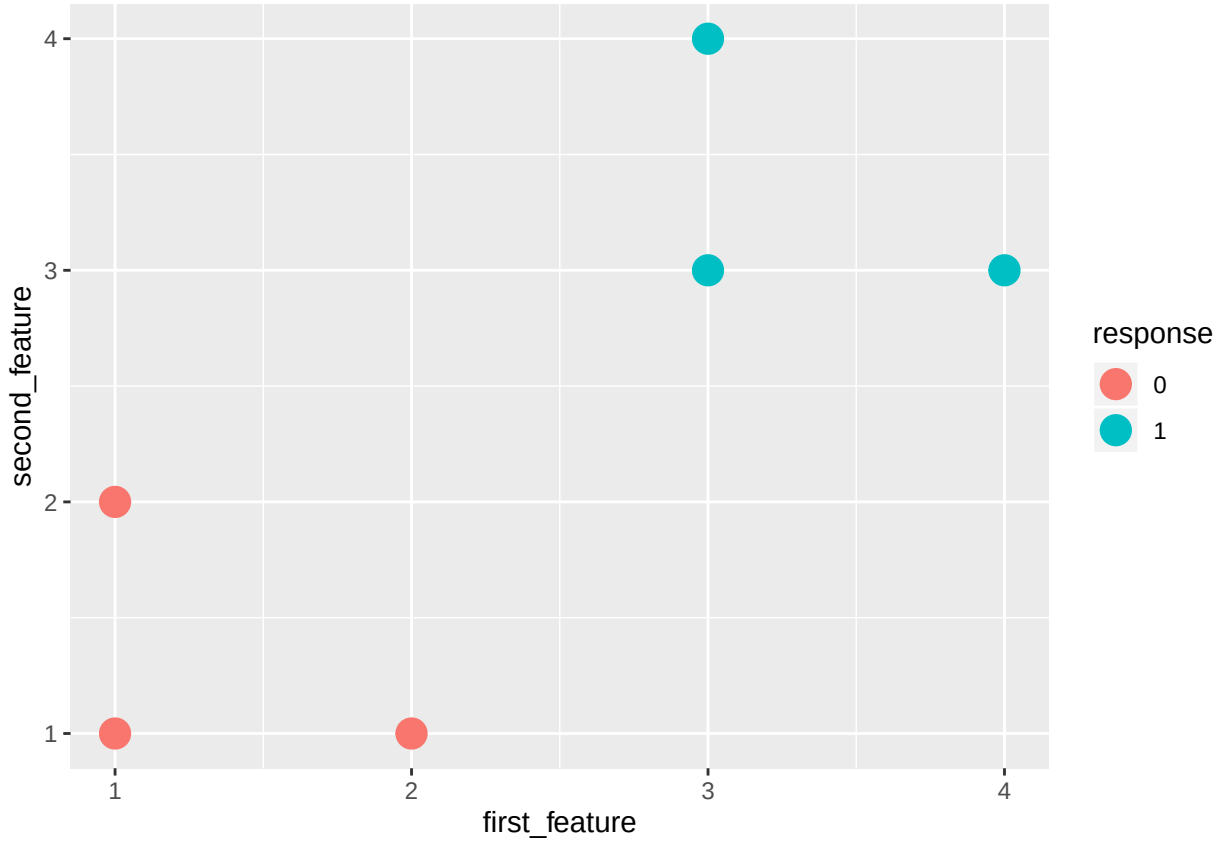
We haven't spoken about visualization yet, but it is important we do some of it now. First we load the visualization library we're going to use:

```
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
```




The scatterplot displays linearly separable data, two classifications of data (0 and 1), with firt_feature on the x-axis and second feature on the y-axis.

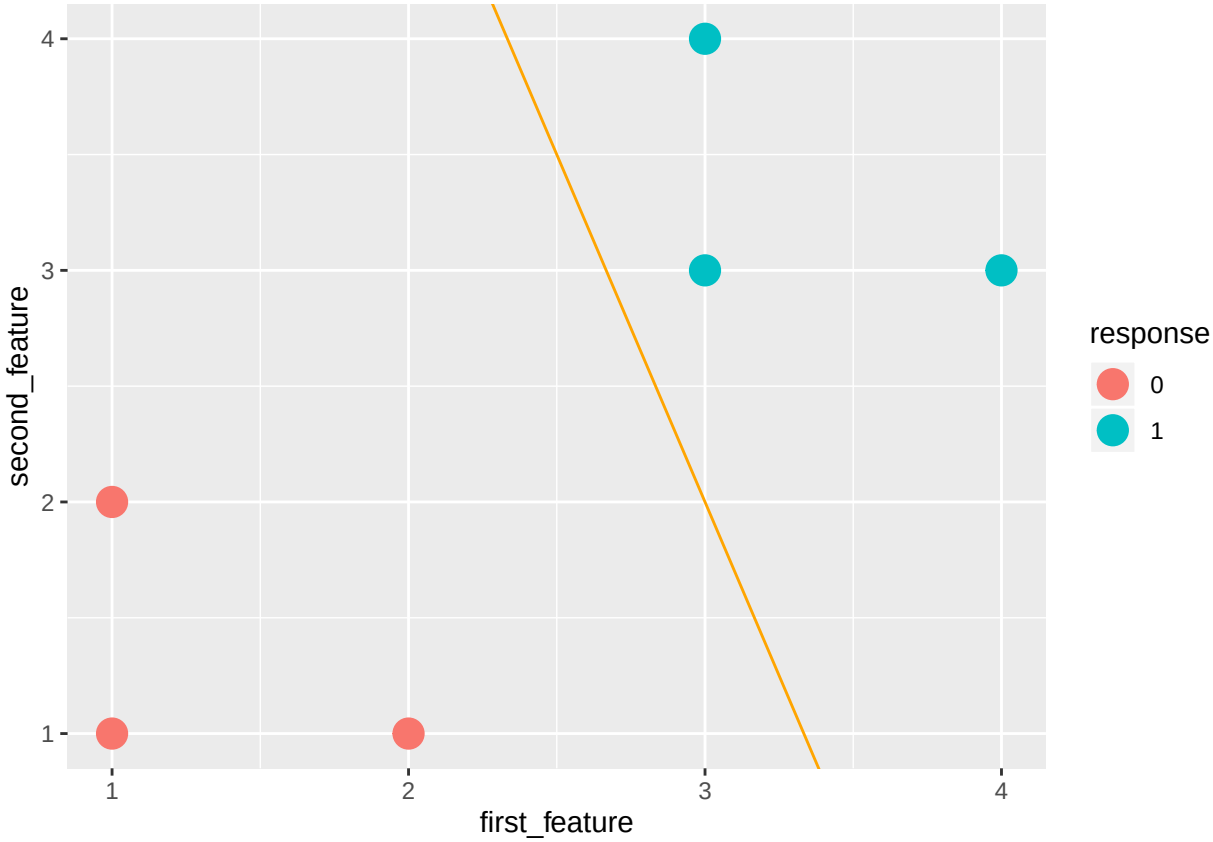
Now, let us run the algorithm and see what happens:

```
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] -11   3   1
```

These numbers are the w0, w1, w2 of the w vector, respectively. The slope is -3 and the intercept is 11.

```
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
```




This is showing the line returned by the perceptron learning algorithm separating two linearly separable sets of data. The line is not satisfying because it is not evenly separated between the two sets (the distance margin is not maximized).

## Support Vector Machine

```
X_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
X_simple_feature_matrix
```

```
##      first_feature second_feature
## [1,]             1              1
## [2,]             1              2
## [3,]             2              1
## [4,]             3              3
## [5,]             3              4
## [6,]             4              3
```

```
y_binary = as.numeric(Xy_simple$response == 1)
y_binary
```

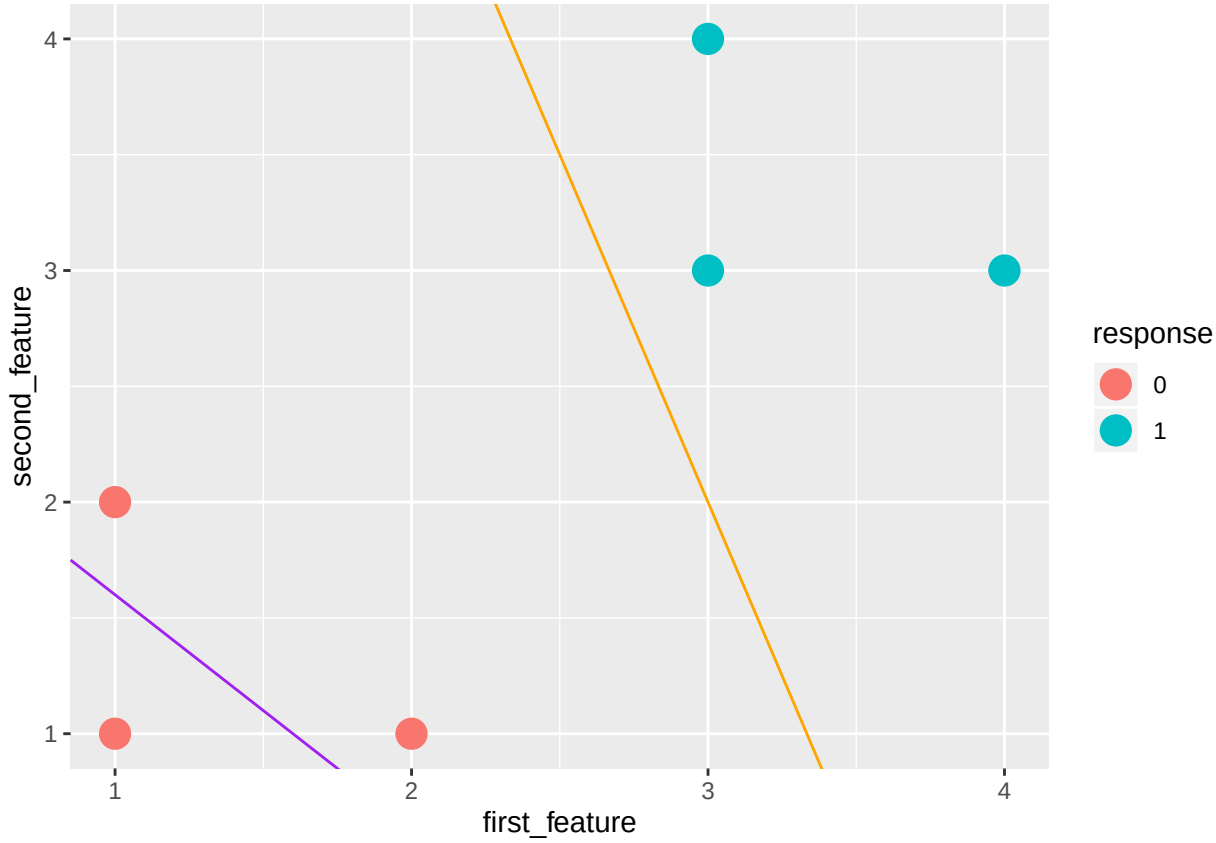
```
## [1] 0 0 0 1 1 1
```

Use the `e1071` package to fit an SVM model to `y_binary` using the features in `X_simple_feature_matrix`. Do not specify the $\lambda$ (i.e. do not specify the `cost` argument). Call the model object `svm_model`. Otherwise the remaining code won't work.

```
pacman::p_load(e1071)
n = nrow(X_simple_feature_matrix)
svm_model = svm(X_simple_feature_matrix, y_binary, 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) %*% X_simple_feature_matrix[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_perceptron_line + simple_svm_line
```




Is this SVM line a better fit than the perceptron?

No, it's not even between the 2 classifications of data.

3. Now write pseuocode for your own implementation of the linear support vector machine algorithm

respecting the following spec making use of the nelder mead `optimx` function from lecture 5p. It turns out you do not need to load the package `neldermead` to use this function. You can feel free to define a function within this function if you wish.

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){

  norm_vec <- function(x){ sqrt(sum(x^2)) }
  pacman::p_load(optimx)
  p = ncol(Xinput)


  compute <- function(w_vec, Xinput = Xinput, y_binary = y_binary, lambda = lambda){

    norm_vec <- function(x){ sqrt(sum(x^2)) }

    Xinput = cbind(1, Xinput)
    b = -w_vec[1] / w_vec[3]
    SHE = 0
    for(i in 1 : nrow(Xinput)){
      max1 = max(0, (0.5 - (y_binary[i] - 0.5)*(w_vec%*%Xinput[i, ] - b)))
      SHE = SHE + max1
      }
    AHE = SHE / n
    AHE + lambda*((norm_vec(w_vec))^2)
  }



  optim_output = optimx(rep(1, p+1), compute, method = "Nelder-Mead")
  w_vec = t(as.matrix(optim_output[1:2]))
}
```

If you are enrolled in 390 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 we discussed in class.

```
#' 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 interce
#     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?

4. Write a $k = 1$ nearest neighbor algorithm using the Euclidean distance function. Respect the spec below:

```
#' This function implements the nearest neighbor algorithm.
#'
#' @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 Xtest       The test data that the algorithm will predict on as a n* x p matrix.
#' @return            The predictions as a n* length vector.
nn_algorithm_predict = function(Xinput, y_binary, Xtest){

#initializes the return vector to NA's
return_Vec = rep(NA, nrow(Xtest))

#Loops through the rows of the test matrix and computes the row index for the
#nearest neighbor in the training data matrix, then returns the binary value associated with
#that row into the return vector
for (t in 1 : nrow(Xtest)){
  best_sqd_distance = Inf
  i_star = NA
  dsqd = 0

  row_Vec_Test = t(as.matrix(Xtest[t, ]))

  for (i in 1 : nrow(Xinput)){
    row_Vec_Input = t(as.matrix(Xinput[i, ]))

    dsqd = sqrt(rowSums((row_Vec_Test - row_Vec_Input)^2))

    if (dsqd < best_sqd_distance){
      best_sqd_distance = dsqd
      i_star = i
    }

  }

  return_Vec[t] = y_binary[i_star]
```

```
}

return_Vec

}
```

Write a few tests to ensure it actually works:

```
X_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
X_simple_feature_matrix
```

```
##      first_feature second_feature
## [1,]             1              1
## [2,]             1              2
## [3,]             2              1
## [4,]             3              3
## [5,]             3              4
## [6,]             4              3
```

```
y_binary = as.numeric(Xy_simple$response == 1)
y_binary
```

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

```
test = rbind(c(3, 2.5), c(3.5, 3.5), c(2, 1.5),  c(1, 1.5), c(7,5))
test
```

```
##      [,1] [,2]
## [1,]  3.0  2.5
## [2,]  3.5  3.5
## [3,]  2.0  1.5
## [4,]  1.0  1.5
## [5,]  7.0  5.0
```

```
nn_algorithm_predict(X_simple_feature_matrix, y_binary, test) #Should return 1, 1, 0, 0, 1
```

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

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.

```
nn_algorithm_predict = function(Xinput, y_binary, Xtest, d = function(A, B){sqrt(rowSums((A - B)^2))}){

#iInitializes the return vector to NA's
return_Vec = rep(NA, nrow(Xtest))

#Loops through the rows of the test matrix and computes the row index for the
#nearest neighbor in the training data matrix, then returns the binary value associated with
#that row into the return vector
for (t in 1 : nrow(Xtest)){
  best_sqd_distance = Inf
  i_star = NA
  dsqd = 0

  row_Vec_Test = t(as.matrix(Xtest[t, ]))

  for (i in 1 : nrow(Xinput)){
```

```
    row_Vec_Input = t(as.matrix(Xinput[i, ]))

    dsqd = d(row_Vec_Test, row_Vec_Input)

    if (dsqd < best_sqd_distance){
      best_sqd_distance = dsqd
      i_star = i
    }
  }
  return_Vec[t] = y_binary[i_star]
}

return_Vec

}
```

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 $\hat{y}$ randomly. Set the default `k` to be the square root of the size of $\mathcal{D}$ 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.

```
nn_algorithm_predict = function(Xinput, y_binary, Xtest, d = function(A, B){sqrt(rowSums((A - B)^2))}, k

#Create function for finding mode
getmode <- function(v) {
   uniqv <- unique(v)
   uniqv[which.max(tabulate(match(v, uniqv)))]
}

#Initializes the return vector to NA's
return_Vec = rep(NA, nrow(Xtest))

#Loops through the rows of the test matrix and computes the row index for the
#nearest neighbor in the training data matrix, then returns the binary value associated with
#that row into the return vector
for (t in 1 : nrow(Xtest)){
  best_sqd_distance = Inf
  i_star = NA
  dsqd = 0

  dsqd_DF = matrix(NA, nrow = nrow(Xinput), ncol = 1)

  row_Vec_Test = t(as.matrix(Xtest[t, ]))

  for (i in 1 : nrow(Xinput)){
    row_Vec_Input = t(as.matrix(Xinput[i, ]))

    dsqd = d(row_Vec_Test, row_Vec_Input)

    dsqd_DF[i, ] = dsqd
  }

  lowest_dsqd_Rows = order(dsqd_DF)[1:k]
```

```
  return_Vec[t] = getmode(y_binary[lowest_dsqd_Rows])
}

return_Vec

}
```