Lab 3

Karen Lopez

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.

```r
#' Perceptron Learning Algorithm
#'
#' TO-DO: Explain what this function does in a few sentences
#'
#' @param Xinput      n x p dimension matrix where n = # of observations & p = # of features
#' @param y_binary    binary vector of size n
#' @param MAX_ITER    the maximum number of iterations
#' @param w           intial weight vector
#'
#' @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            [Karen Lopez]
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w = NULL){
  Xinput = cbind(1, Xinput)
  p = ncol(Xinput)
  n = nrow(Xinput)

  if(is.null(w)){
    w = runif(ncol(Xinput))
  }

  for(iter in 1 : MAX_ITER){
    for(i in 1 : nrow(Xinput)){
      x_i = Xinput[i, ]
      y_hat_i = ifelse(sum(x_i * w) > 0 , 1, 0)
      w = w + as.numeric(y_binary[i] - y_hat_i) * x_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 $\mathbb{D}$.

```r
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
)
```

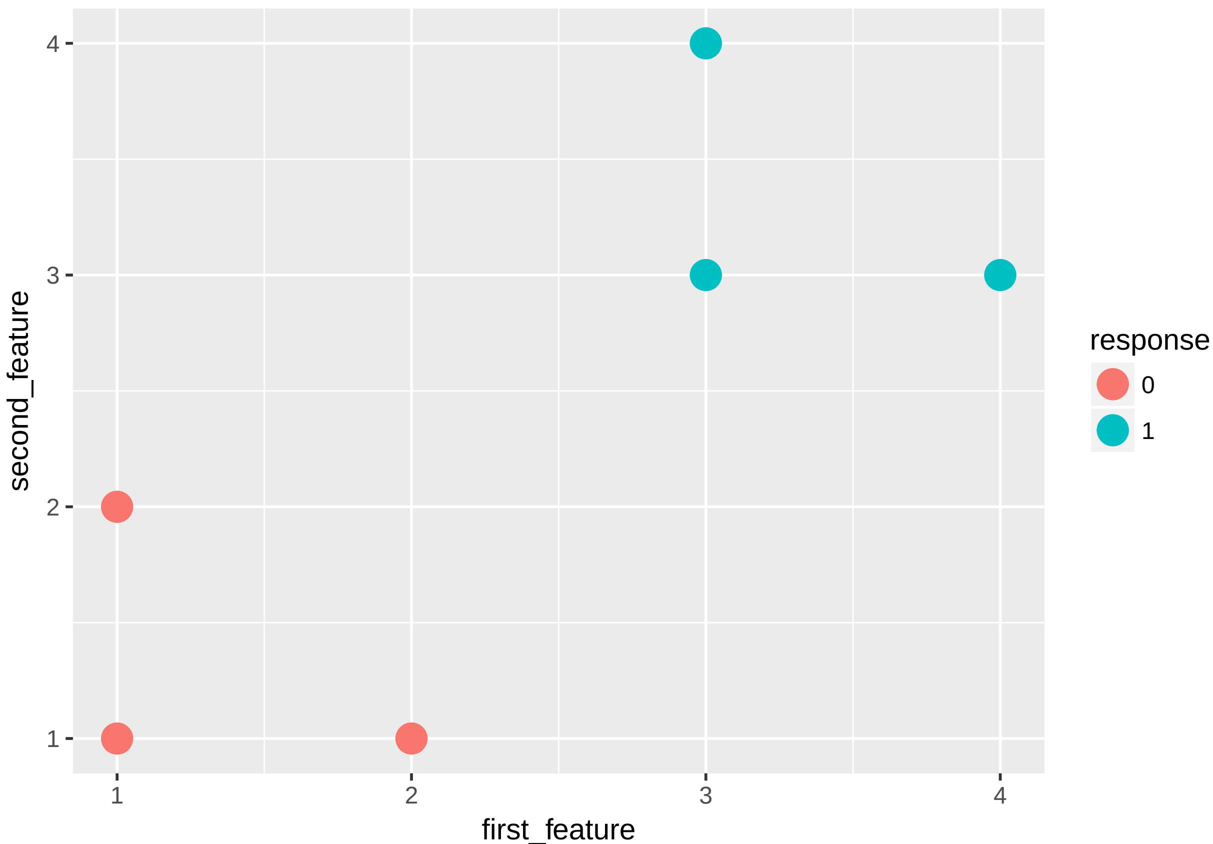
We haven't spoken about visualization yet, but it is important we do some of it now. First we load the visualization library we're going to use:

```
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
```



THis picture visualizes the binary data of 0(red) and 1's (blue)

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

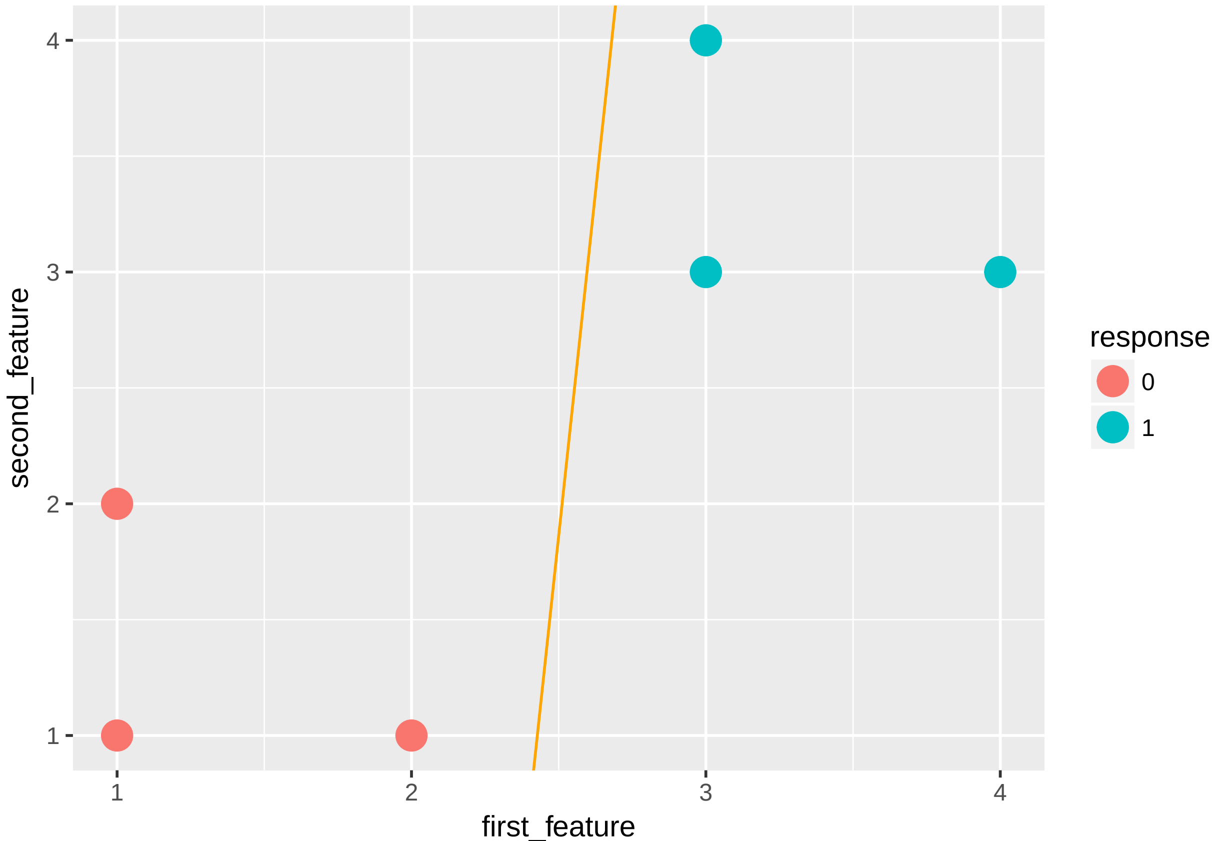
```
w_vec_simple_per = perceptron_learning_algorithm(
  cbind(Xy_simple$first_feature, Xy_simple$second_feature), #merges by columns
  as.numeric(Xy_simple$response == 1))
w_vec_simple_per
```

```
## [1] -1.05059942  0.44836524 -0.03777355
```

TO-DO: 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. provides the w_vec for the perceptron learning algorithm. Intercept: -24.29 Slope: 8.39

```
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
```



TO-DO: Explain this picture. Why is this line of separation not "satisfying" to you? It does not create a maximum wedge.

Support Vector Machine

```
X_simple_feature_matrix = as.matrix(Xy_simple[, 2 : 3])
y_binary = as.numeric(Xy_simple$response == 1)
```

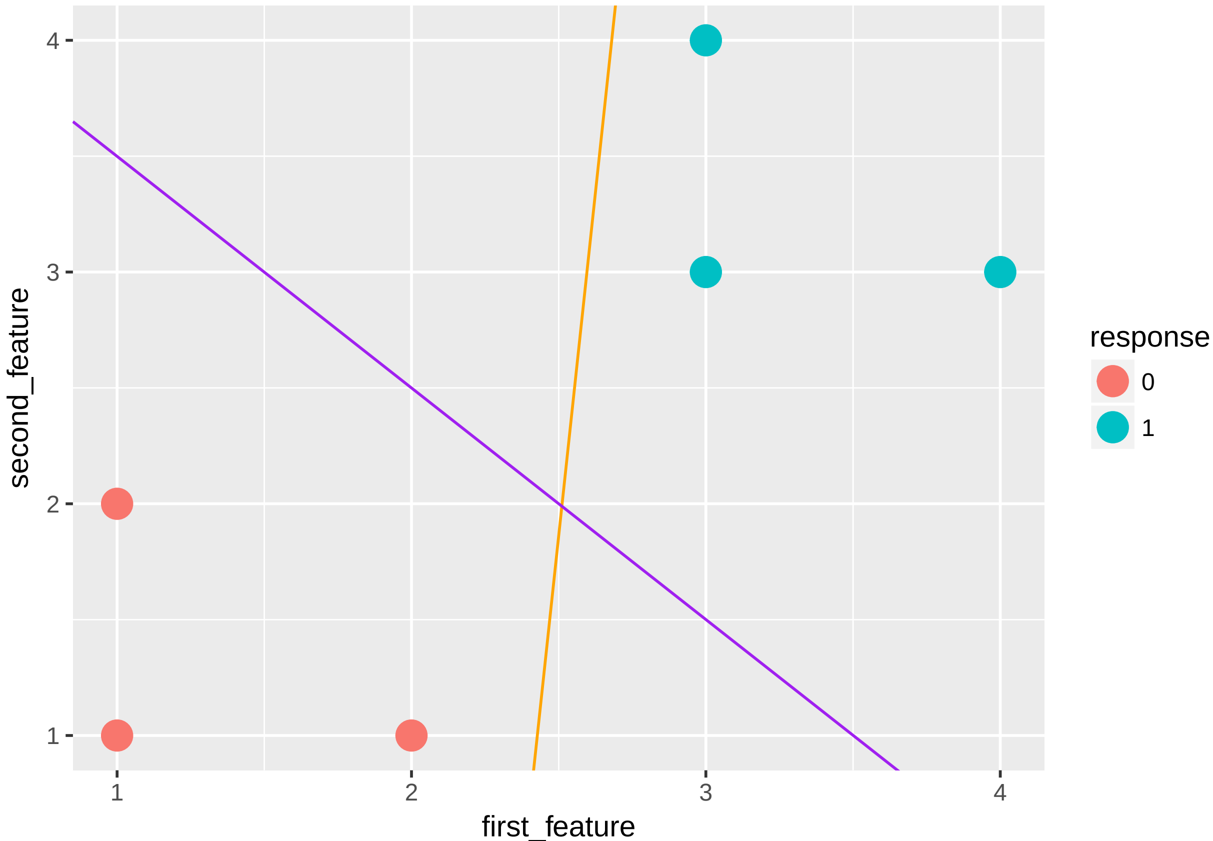
Use the `e1071` package to fit an SVM model to `y_binary` using the features in `X_simple_feature_matrix`. Do not specify the λ (i.e. do not specify the `cost` argument). Call the model object `svm_model`. Otherwise the remaining code won't work.

```
pacman::p_load(e1071)
svm_model = svm(X_simple_feature_matrix , Xy_simple$response, 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? The SVM line is a better fit because it offers a more oprtimal wedge.

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){
  #defining p features of n calums of Xinput
  #Xinput of a combination of 1s
```

```
  #
}
```

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 intercep
#    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){
 return = rep(NA, nrow(Xtest))
 bestSquaredDistance = Inf
 i_star = NA
  for (i in 1:nrow(Xtest) ){
    for(index in 1:nrow(Xinput)){
      eucl = sqrt(sum((Xinput[i, ] - Xtest[i, ])^2))
      if (eucl < bestSquaredDistance){
        bestSquaredDistance = eucl
        iStar = i
        return[i] = y_binary[iStar]

      }
    }
    bestSquaredDistance=Inf
  }
```

```
 return
}
```

Write a few tests to ensure it actually works:

```
#TO-DO
```

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.

```
euclidean_metric <- function(x_1, x_2){
  ((x_1 - x_2)^2)
}
nn_algorithm_predict_d = function(Xinput, y_binary, Xtest, d=euclidean_metric){
prediction = c(rep(NA , nrow(Xtest)))
iStar = c(rep(NA, nrow(Xtest)))
for(k in 1 : nrow(Xtest)){
best_sqd_distance = Inf
for(i in 1 : nrow(Xinput)){
total_dsqrd = 0
for (j in 1 : ncol(Xinput)) {
dsqd = euclidean_metric(Xinput[i,j], Xtest [k,j])
total_dsqrd = total_dsqrd + dsqd
}
if(dsqd< best_sqd_distance){
best_sqd_distance = dsqd
iStar[k]= i
}
}
prediction[k]= y_binary[iStar [k]]
}
prediction
}
```

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.

```
#TO-DO --- extra credit for undergrads
```