Lab 3

Moshe Weiss

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
#' Fits a line between data, assuming linear separability
#' @param Xinput      Numeric matrix of dim(nxp) where n is number of observations and p is number of f
#' @param y_binary    Binary vector of length n
#' @param MAX_ITER    Number of iterations, defaults to 1000
#' @param w           Initial numeric vector of weights 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            [your name]
perceptron_learning_algorithm = function(Xinput, y_binary, MAX_ITER = 1000, w = NULL){
  n=nrow(Xinput)
  Xinput = cbind(rep(1,n),Xinput)
  p=ncol(Xinput)

  if(is.null(w)){
    w=rep(0,p) #zero vector
  }
  for(iter in 1:MAX_ITER){
    for(i in 1:n){
      currentX = Xinput[i,]
      delta_y_i = y_binary[i] - ifelse(sum(currentX*w)>0,1,0)
      for(j in (1:p)){
        w[j] = w[j]+delta_y_i*currentX[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
```
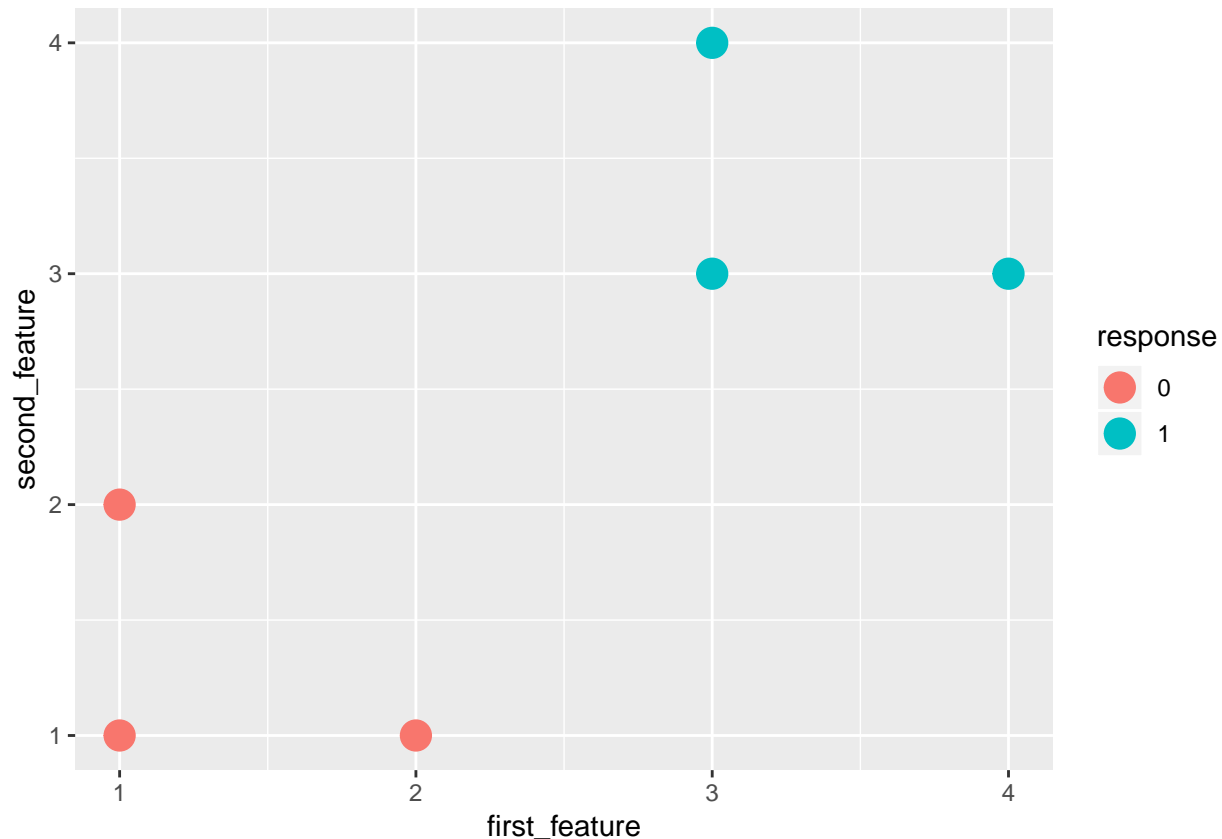

These are the data points plotted on a plane where the x axis is the first feature and the y axis is the second feature. They are color-coded based on classification.

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

```
Xy_simple$first_feature
```

```
## [1] 1 1 2 3 3 4
```

```
w_vec_simple_per = perceptron_learning_algorithm(
  cbind(Xy_simple$first_feature, Xy_simple$second_feature),
  as.numeric(Xy_simple$response == 1),1000)
w_vec_simple_per
```

```
## [1] -7  4 -1
```

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.

Intercept = -7 (bias, offset) Slope = 4 (for an increase in value of x1 by 1, y will have an g(x) will increase by 4) Unsure how to reconcile b1 and b2 in terms of slope, but in the code snippet below, intercept and slope are both specified.

```
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
```
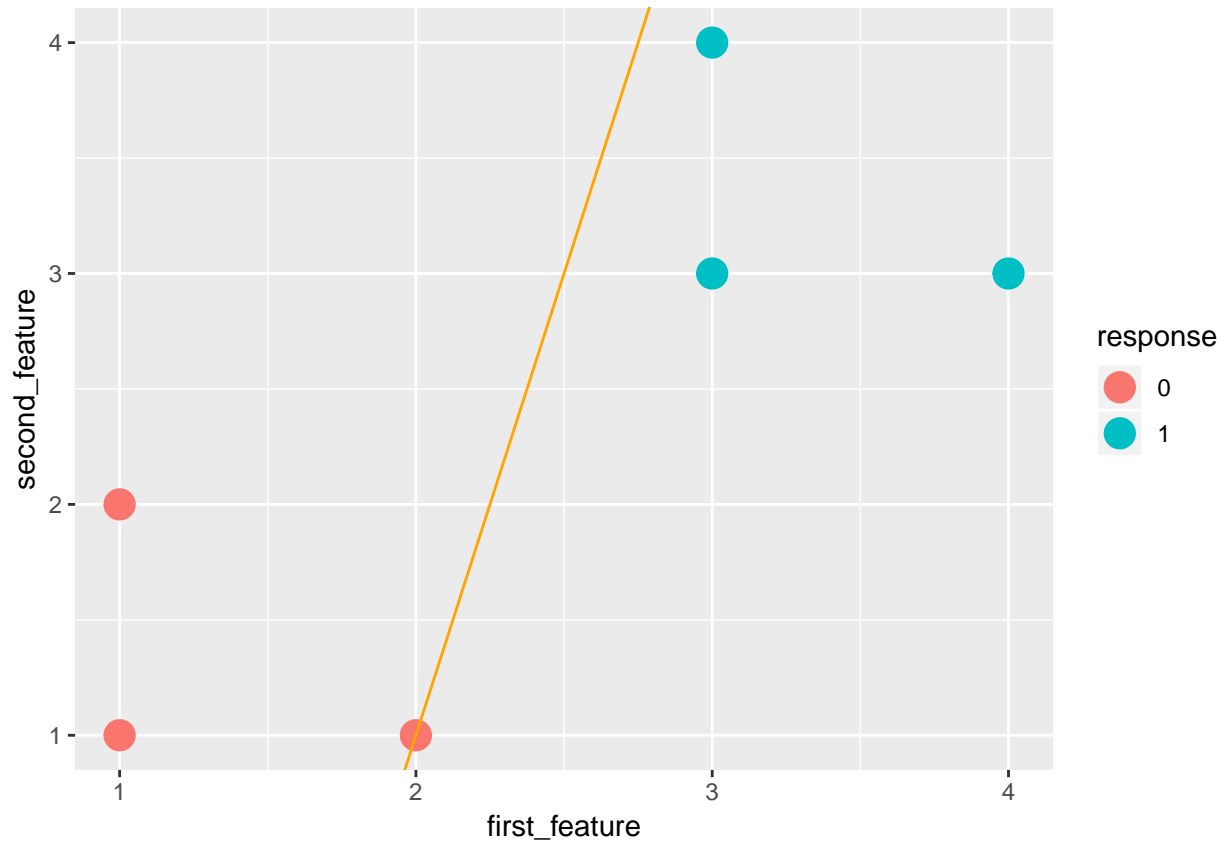

Explain this picture. Why is this line of separation not "satisfying" to you?

Answer: The alignment of dots almost beg to have a line of negative slope. In terms of the SVM, this line strictly separates; it clearly does not create a maximum wedge.
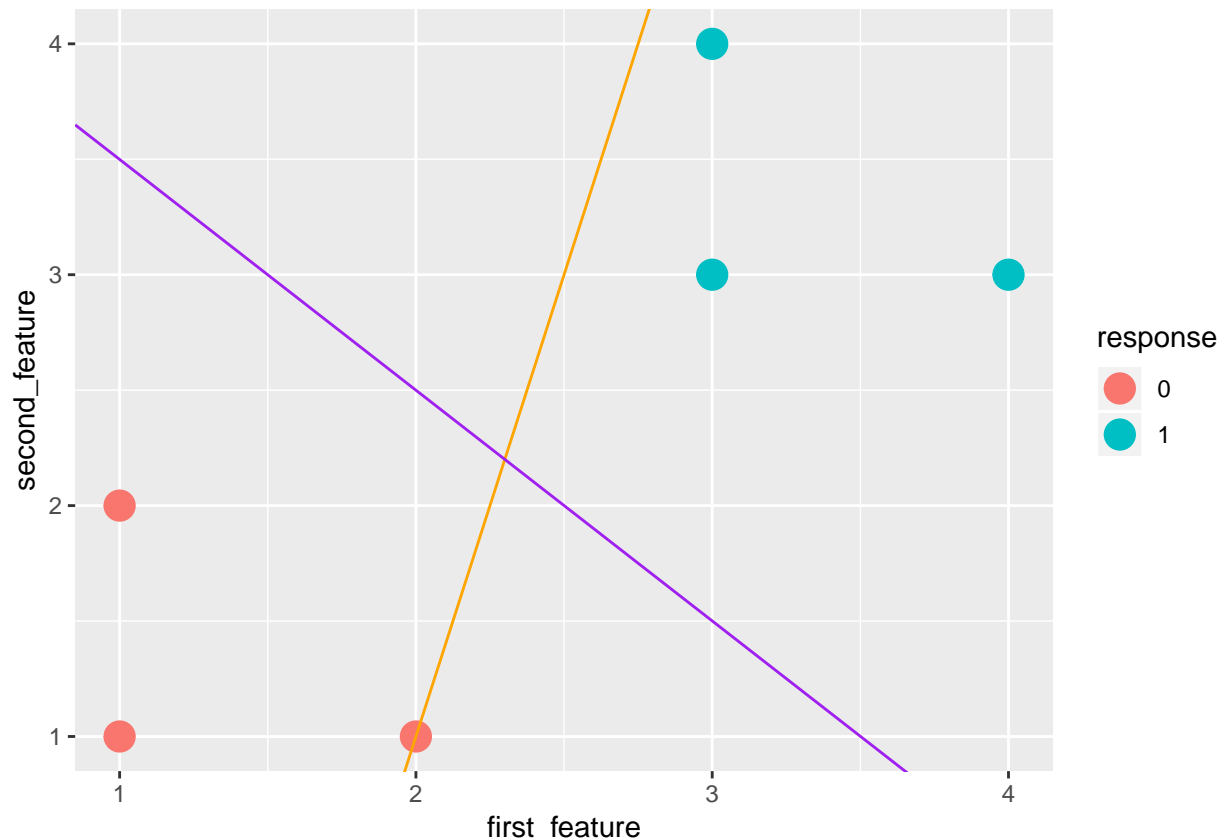
Support Vector Machine

```
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=NULL)
```

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

3

```
?optim
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?

Yes

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){
  #AvgHingeError = sum(max((y_i-0.5)*(w_vec%*%Xinput+b),0))/n
  #sltn = min(AHE + lambda*magnitude(w_vec)^2)
  #sltn
}
```

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){
  n = nrow(Xinput)
  distances = c()
  min = 1
  for(i in (1:n)){
    distances = c(distances, sum((Xinput[i,]-Xtest)^2))
    if(distances[i]<distances[min]) min = i
  }
  y_binary[min]
}
```

Write a few tests to ensure it actually works:

```
nn_algorithm_predict(X_simple_feature_matrix,y_binary,c(1,2))
```

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

```
#closest to (1,2), expect 0
nn_algorithm_predict(X_simple_feature_matrix,y_binary,c(1,1))
```

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

```
#closest to (1,1), expect 0
nn_algorithm_predict(X_simple_feature_matrix,y_binary,c(3,2))
```

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

```
#closest to (3,3), expect 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.

```
#' 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.
#' @param distFunction The distance function that the algorithm uses. Defaults to the Euclidean Distanc
```

```
#' @return              The predictions as a n* length vector.
nn_algorithm_predict = function(Xinput,
                                y_binary,
                                Xtest,
                                distFunction=function(x, xhat){sum((x-xhat)^2)}
                                )
  {
    n = nrow(Xinput)
    distances = c()
    min = 1
    for(i in (1:n)){
      distances = c(distances, distFunction(Xinput[i,],Xtest))
      if(distances[i]<distances[min]) min = i
  }
  y_binary[min]
}
```

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.

```
#' 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.
#' @param distFunction The distance function that the algorithm uses. Defaults to the Euclidean Distanc
#' @param k            The number of nearest neighbors to select. default is 1
#' @return             The predictions as a n* length vector
nn_algorithm_predict = function(Xinput,
                                y_binary,
                                Xtest,
                                distFunction=function(x, xhat){sum((x-xhat)^2)},
                                k=1
                                )
  {
    n = nrow(Xinput)
    distances = c()
    min = 1
    neighbors = array(dim=k)
    for(i in (1:n)){
      neighbors = sort(neighbors,na.last = FALSE)
      distances = c(distances, distFunction(Xinput[i,],Xtest))
      if(distances[i]<distances[min]) min = i
      for(j in (1:k)){
        if(is.na(neighbors[j])){
          neighbors[j]=i
          break
        }else if(distances[neighbors[j]]>distances[i]){
          neighbors[j] = i
          break
        }
```

```r
    }
  }
  if(sum(y_binary[neighbors])>k/2){
    1
  }else if(sum(y_binary[neighbors])<k/2){
    0
  }else{
    sample(c(0,1),1)
  }
}
#This is assuming a binary classification
```