Labsheet - 8

Perceptron

Machine Learning

BITS F464

I Semester 2021-22

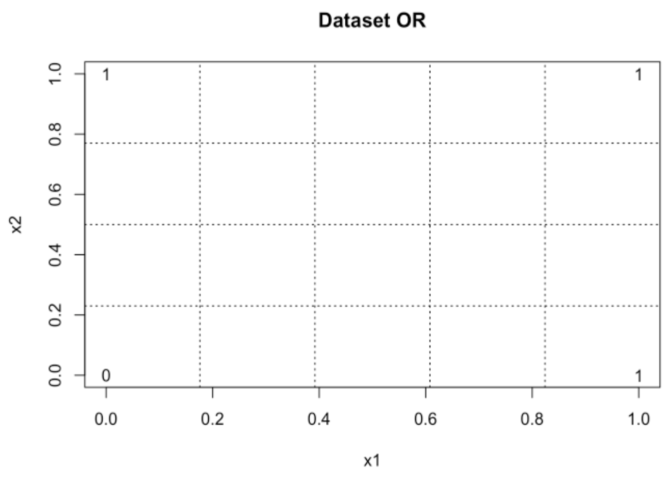
Perceptron

We are going to model the well known OR logic function using perceptron.

```
data = data.frame( x1 = c( 0, 0, 1, 1 ),
                   x2 = c( 0, 1, 0, 1 ),
                   y =  c( 0, 1, 1, 1 ) )

print( data )
```

The dataset has only four observations which two of it belongs to class 0 and others to class 1. It is worth to see data in a graph.

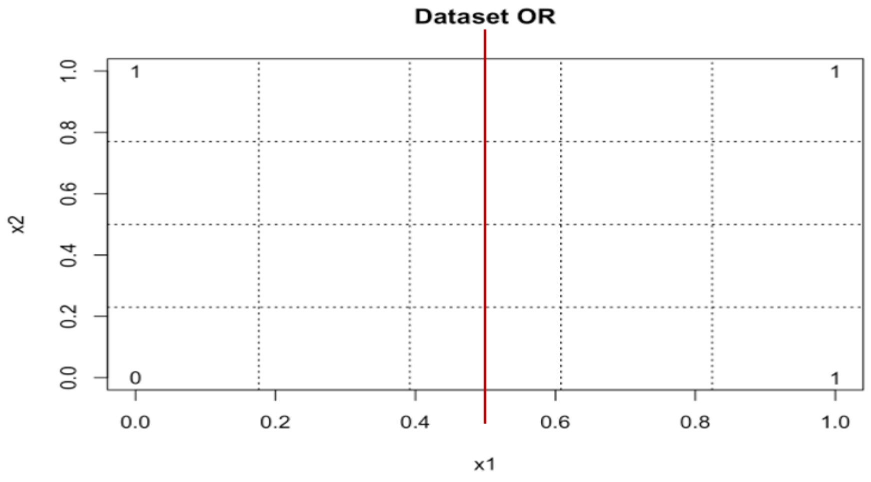
```
plot( data$x1, data$x2, type = 'n', main = 'Dataset OR', xlab = "x1",
ylab = "x2" )
text( data$x1, data$x2, labels = data$y )
grid( nx = length( data$x1 ) + 1, ny = length( data$x1 ), col = 'black
' )
```



Weight Initialization

The initial solution is X_1=0.5

```
weights =  c( -0.5, 1, 0 )
```



We can also initialize the weights randomly within a short range of values, like this:

```
weights = rnorm( mean = 0, sd = 0.1, n = ncol( data ) )

print( weights )
```

I took numbers randomly from normal distribution with mean equal zero and standard deviation equal 0.1. Wait a minute, so far we have seen two weights, one for each input, however the code above shows three weights. The third weights are for the bias inputs. Bias is a fixed input, generally either -1 or 1 that is responsible to activate a neuron even if the current inputs are both zeros. The last definition missing is the activation function.

The property of Heaviside step function is to have output equal 0 when the input is below a certain threshold and output 1 otherwise. You can already have the feel that the output of an activation function will be the guessing of the algorithm about what class the input belongs to. In this scenario, the Heaviside step function makes sense, because it allows us to compute the error in order to evaluate the performance of the classifier. Considering the value of the threshold equal 0.5, the activation function can be written as follows:

```
activation_function = function( net ){
                        if( net > 0.5 )
                            return( 1 )
                        return( 0 )
                    }
```

Now, we can finally write down the code for perceptron. Let’s take the first observation, add the bias values and multiply by the initial weights, the result will be put in activation function. After this, the value of error is computed and the weights are update in order to improve the assertiveness. To compute the error and update the weights, we are going to use the following expression where the parameter eta is the learning rate, this value define the step to walk throughtout the error function.

```
data = as.matrix( data )
net = c( data[1, 1:2 ], 1 ) %*% weights

y_hat = activation_function( net )

error = y_hat - data[1,3]

eta =  0.1

weights = weights - eta * ( error ) * c( data[ 1, 1:2 ], 1 )

print( weights )
```

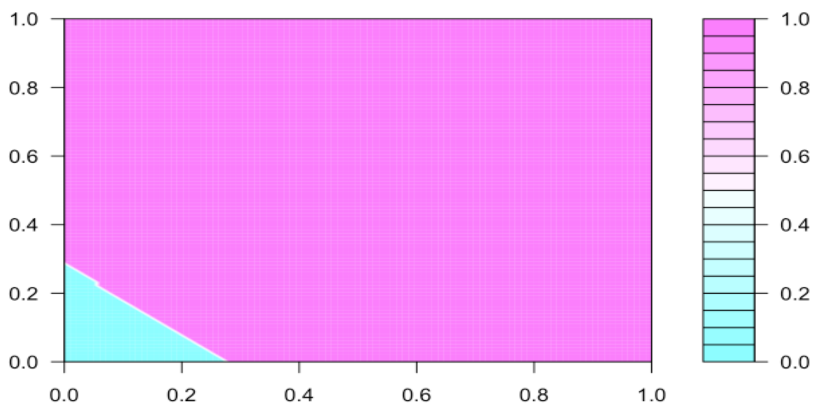
This is the learning process with only the first observation, now we need to repeat it for the all examples available on the dataset. However, this loop needs to have a stop criterion which will define when the learning process should stop. To measure performance of the algorithm in classification task, we are observing the value of error, so it sounds reasonable to use it as a stop criterion, to be more precisely we are going to use the mean square error (mse) value. When the algorithm reaches a low value of mse, we will be satisfied with its performance. The code that performs the entire workflow training can be seen below:

```
perceptron = function( dataset, eta = 0.1, threshold = 1e-5 ){
                        data = as.matrix( dataset )
                        num.features = ncol( data ) - 1
                        target = ncol( data )

                        # Initial random  weights
                        weights = rnorm( mean = 0, sd = 0.1, n = ncol(
data ) )


                        mse = threshold * 2
                        while( mse > threshold ){
                            mse = 0
                            for( i in 1:nrow( data ) ){
                                # Add bias and compute multiplications
                                net = c( data[ i, 1:num.features ], 1
) %*% weights


                                # Activation function
                                y_hat = activation_function( net )
                                # Compute mse
                                error = ( y_hat - data[ i, target ] )
                                mse = mse + error^2
                                cat( paste( "Mean square error = ", ms
e, "\n" ) )

                                # Update weights
                                weights = weights - eta * error * c( d
ata[i, 1:num.features ], 1 )
                            }
                        }
                        return( weights )
                    }
```

After we find the optimal weights for perceptron model, let's take a look at the hyperplane. The function below charts the hyperplane over the input space.

```
shattering.plane = function( weights ){
                        X = seq( 0, 1, length = 100 )
                        data = outer( X, X, function( X, Y ){ cbind( X
, Y, 1 ) %*% weights } )
                        id = which( data > 0.5 )
                        data[ id ] = 1
                        data[ -id ]= 0
                        filled.contour( data )
                    }
```

```
weights = perceptron( data, eta=0.1, threshold=1e-5 )
shattering.plane( weights )
```



As you can see, the hyperplane separates the input space in two regions (blue and pink), each region represents that space where inputs are classified as belonging to class 0 and class 1. Also, you can observe that each time that you run the codes, you will get a different value of weights and consequently a change of the position of the hyperplane.

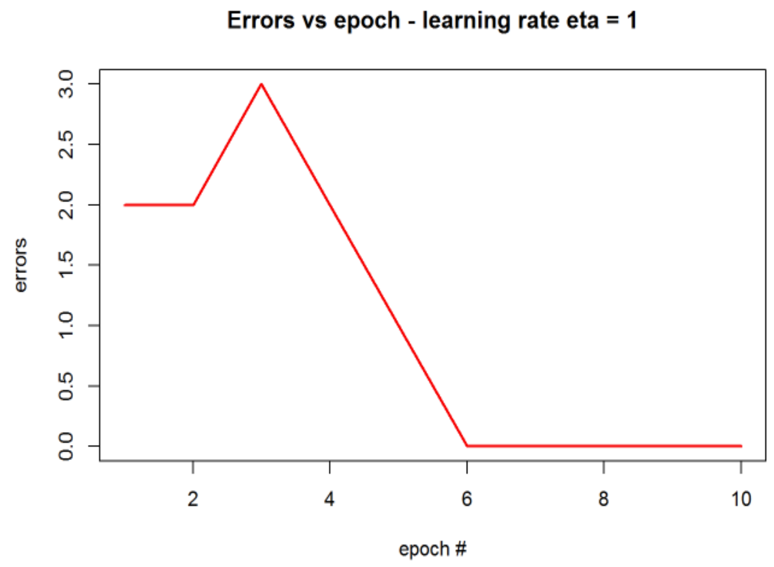
Perceptron - Binary classification algorithm

we use the Iris data set.

```
# load iris data set
data(iris)

# subset of iris data frame - extract only species versicolor and seto
sa
# we will only focus on the sepal and petal lengths of the dataset
irissubdf <- iris[1:100, c(1, 3, 5)]
names(irissubdf) <- c("sepal", "petal", "species")
head(irissubdf)
```

```
# plot data
library(ggplot2)
ggplot(irissubdf, aes(x = sepal, y = petal)) +
        geom_point(aes(colour=species, shape=species), size = 3) +
        xlab("sepal length") +
        ylab("petal length") +
        ggtitle("Species vs sepal and petal lengths")

# add binary labels corresponding to species - Initialize all values to 1
# add setosa label of -1. The binary +1, -1 labels are in the fourth
# column. It is better to create two separate data frames: one containing
# the attributes while the other contains the class values.
irissubdf[, 4] <- 1
irissubdf[irissubdf[, 3] == "setosa", 4] <- -1

x <- irissubdf[, c(1, 2)]
y <- irissubdf[, 4]

# head and tail of data
head(x)

perceptron <- function(x, y, eta, niter) {

        # initialize weight vector
        weight <- rep(0, dim(x)[2] + 1)
```

```
    errors <- rep(0, niter)


    # loop over number of epochs niter
    for (jj in 1:niter) {

        # loop through training data set
        for (ii in 1:length(y)) {

            # Predict binary label using Heaviside activation
            # function
            z <- sum(weight[2:length(weight)] *
                        as.numeric(x[ii, ])) + weight[1]
            if(z < 0) {
                ypred <- -1
            } else {
                ypred <- 1
            }

            # Change weight - the formula doesn't do anything
            # if the predicted value is correct
            weightdiff <- eta * (y[ii] - ypred) *
                c(1, as.numeric(x[ii, ]))
            weight <- weight + weightdiff

            # Update error function
            if ((y[ii] - ypred) != 0.0) {
                errors[jj] <- errors[jj] + 1
            }

        }
    }

    # weight to decide between the two species
    print(weight)
    return(errors)
}

err <- perceptron(x, y, 1, 10)

plot(1:10, err, type="l", lwd=2, col="red", xlab="epoch #", ylab="errors")
title("Errors vs epoch - learning rate eta = 1")
```



Note that increasing the learning rate does exactly as its name suggests, the only problem being that the rate might not be sensitive enough to ensure convergence. While convergence is ensured, a larger learning rate (between 0 and 1) leads to a faster convergence.

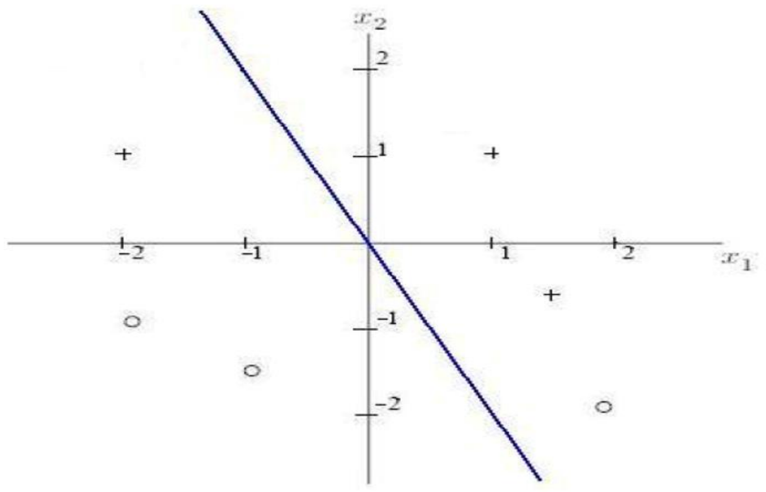
Exercise

- How to figure out the optimum learning rate from the data?
- Implement perceptron algorithm on AND and NAND logic function's sample data.
- Try to generate the data set shown in the below figure. Take initials weights mentioned below with figure and use perceptron model for weights learning. Also, try to change the node output for w_0 and compare the results. Note :(node output value can be -1,1 & 2)

$$\eta = 0.2$$

$$w = \begin{pmatrix} 0 \\ 1 \\ 0.5 \end{pmatrix}$$

$$w_0 = w_0 - 0.2 * 1$$
$$w_1 = w_1 - 0.2 * 2$$
$$w_2 = w_2 - 0.2 * (-2)$$


x_2
2
1
-1
-2
-2
-1
1
2
x_1