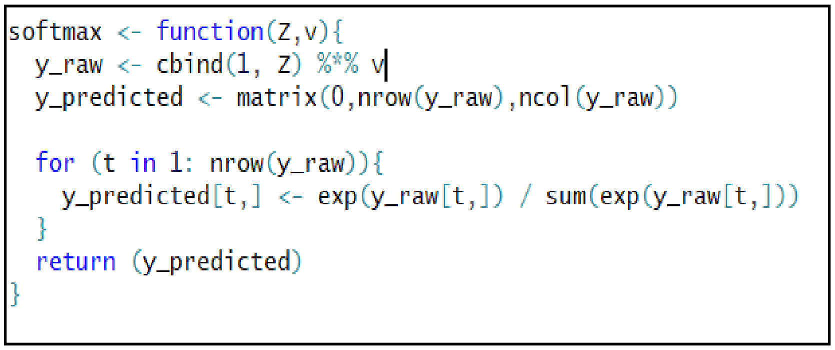
# INDR421 HW#3, Prepared by Arda Kırkağaç

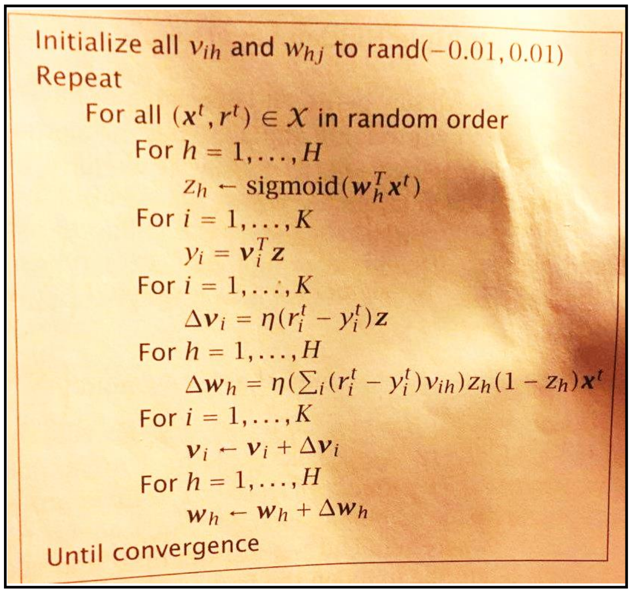
In this homework, we implemented a multiclass multilayer perceptron algorithm. I again used R platform. In data generation part, points from 4 different classes -while having partial distributions over the input space- were generated according to the given means and variances. One-of-K encoding is the main method here, so again generated classes were distributed over 4 different columns, making the matrix a binary one. Then, the points are plotted for visualization.



```
softmax <- function(Z,v){
  y_raw <- cbind(1, Z) %*% v
  y_predicted <- matrix(0,nrow(y_raw),ncol(y_raw))

  for (t in 1: nrow(y_raw)){
    y_predicted[t,] <- exp(y_raw[t,]) / sum(exp(y_raw[t,]))
  }
  return (y_predicted)
}
```

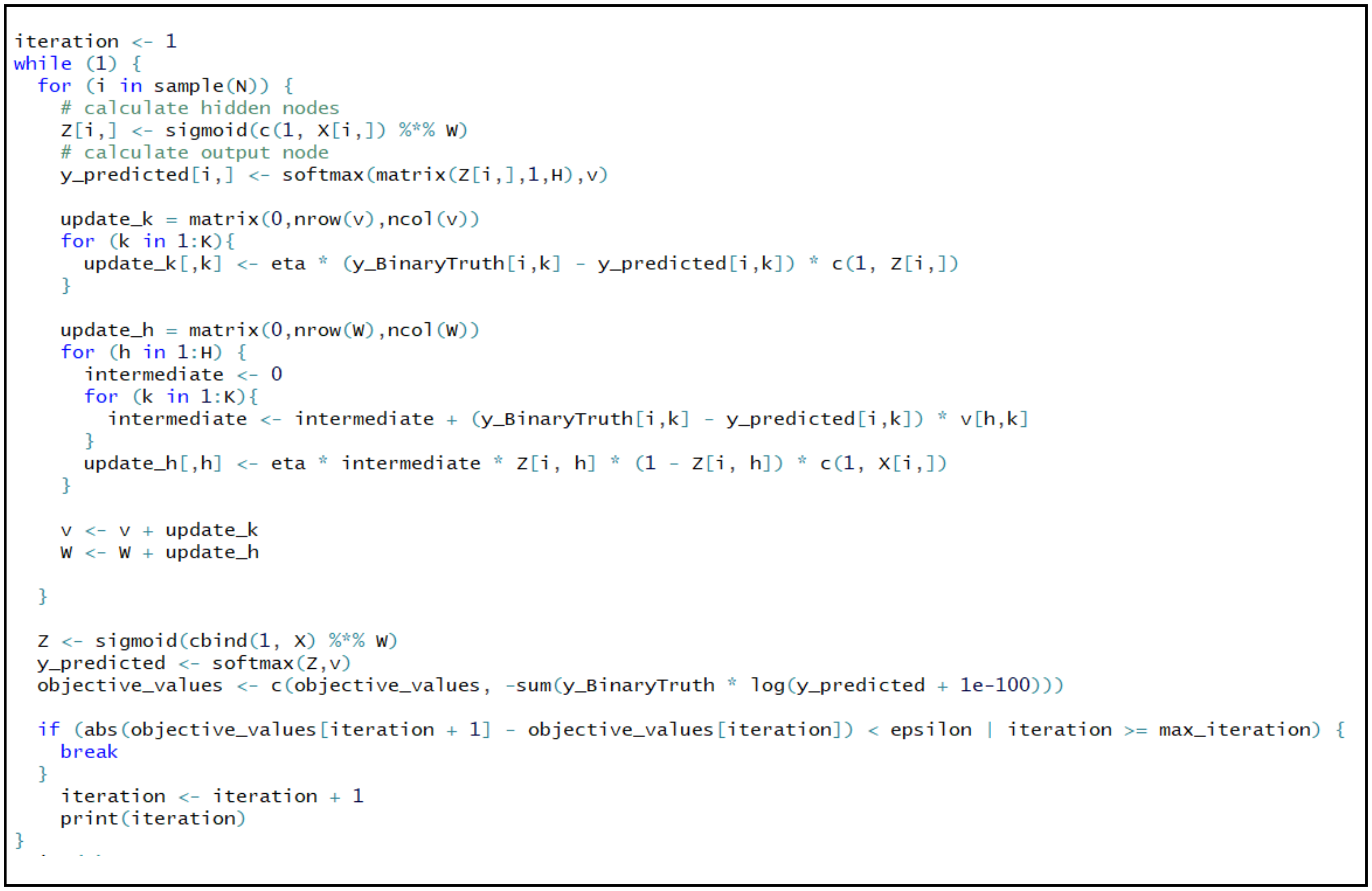
Softmax function is one of the primary functions of this procedure. On the right, you can see my implementation of softmax. It basically takes Z (middle layer values) and v (weights between the hidden layer and the output raw layer) as inputs, and calculates the matrix multiplication out of it. Then for each column discussed above, exponential ratios (softmax values) are calculated. The resulting matrix is basically softmax values for each input value. The dimensions of this matrix are 400*4 in our case, 400 being the number of points and 4 being the number of classes.

The next step was to initialize the learning parameters exacly given as in the assignment document. Then the random weight were initialized. I did not use the second seed 521 this time, since I realized the output is better (this is completely by chance). Then using this values, initial values for estimations and objective function are calculated.



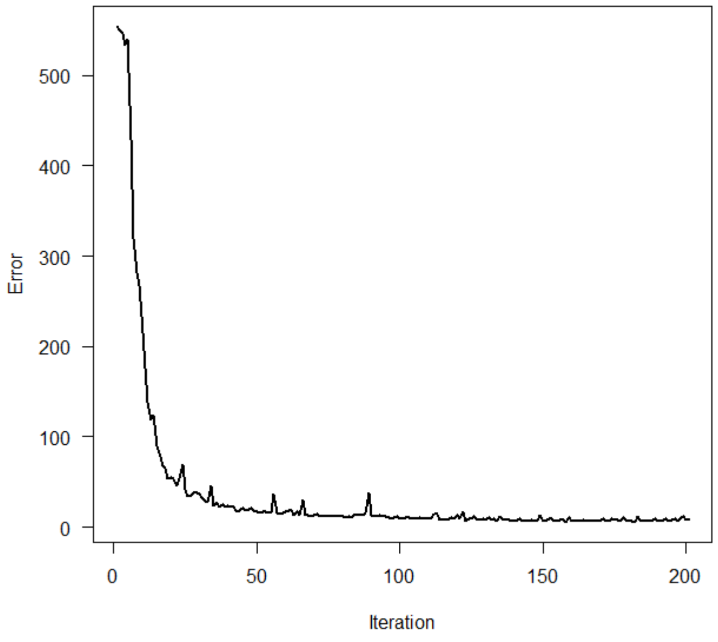
```
Initialize all $v_{ih}$ and $w_{hj}$ to rand$(-0.01, 0.01)$
Repeat
    For all $(x^t, r^t) \in \mathcal{X}$ in random order
        For $h = 1, \ldots, H$
            $z_h \leftarrow \text{sigmoid}(w_h^T x^t)$
        For $i = 1, \ldots, K$
            $y_i = v_i^T z$
        For $i = 1, \ldots, K$
            $\Delta v_i = \eta(r_i^t - y_i^t) z$
        For $h = 1, \ldots, H$
            $\Delta w_h = \eta(\sum_i (r_i^t - y_i^t) v_{ih}) z_h (1 - z_h) x^t$
        For $i = 1, \ldots, K$
            $v_i \leftarrow v_i + \Delta v_i$
        For $h = 1, \ldots, H$
            $w_h \leftarrow w_h + \Delta w_h$
Until convergence
```

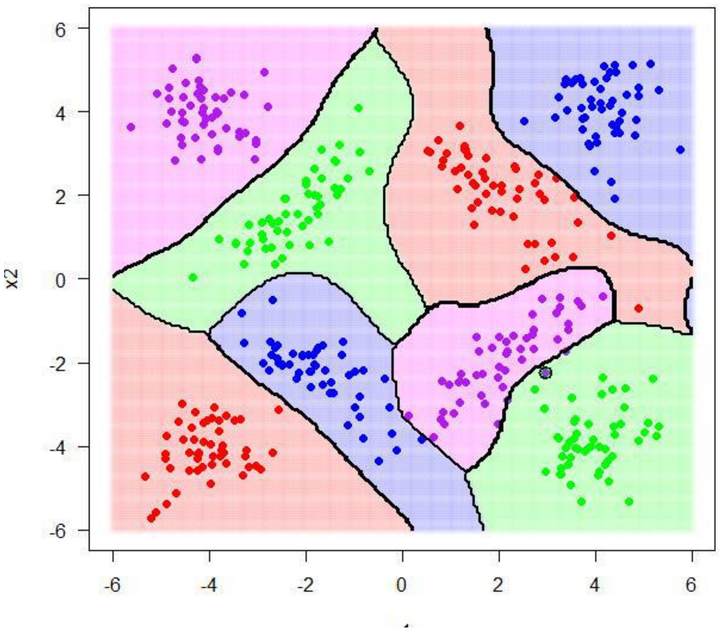
In the iteration, the pseudocode on the right was taken into consideration. For each point in our sample input, Z and y values are calculated. Then, gradient values for the weights between hidden and output layer, and between input and hidden layer are computed respectively. Since gradient expression of W includes the value of v's, the update part are left to the end on purpose. I found out that this method gives a smoother objective plot over iterations, and the classification is a little bit better.

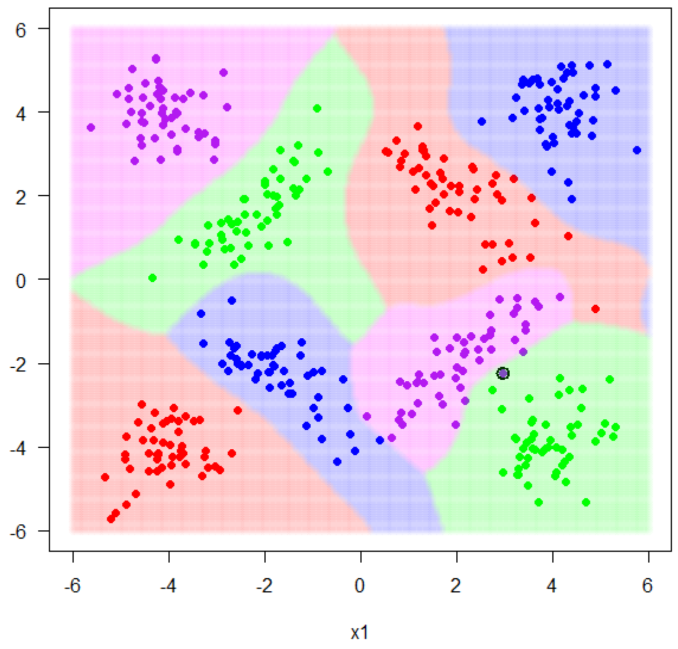


```
iteration <- 1
while (1) {
  for (i in sample(N)) {
    # calculate hidden nodes
    Z[i,] <- sigmoid(c(1, X[i,]) %*% W)
    # calculate output node
    y_predicted[i,] <- softmax(matrix(Z[i,],1,H),v)

    update_k = matrix(0,nrow(v),ncol(v))
    for (k in 1:K){
      update_k[,k] <- eta * (y_BinaryTruth[i,k] - y_predicted[i,k]) * c(1, Z[i,])
    }

    update_h = matrix(0,nrow(W),ncol(W))
    for (h in 1:H) {
      intermediate <- 0
      for (k in 1:K){
        intermediate <- intermediate + (y_BinaryTruth[i,k] - y_predicted[i,k]) * v[h,k]
      }
      update_h[,h] <- eta * intermediate * Z[i, h] * (1 - Z[i, h]) * c(1, X[i,])
    }

    v <- v + update_k
    W <- W + update_h

  }

  Z <- sigmoid(cbind(1, X) %*% W)
  y_predicted <- softmax(Z,v)
  objective_values <- c(objective_values, -sum(y_BinaryTruth * log(y_predicted + 1e-100)))

  if (abs(objective_values[iteration + 1] - objective_values[iteration]) < epsilon | iteration >= max_iteration) {
    break
  }
    iteration <- iteration + 1
    print(iteration)
}
```

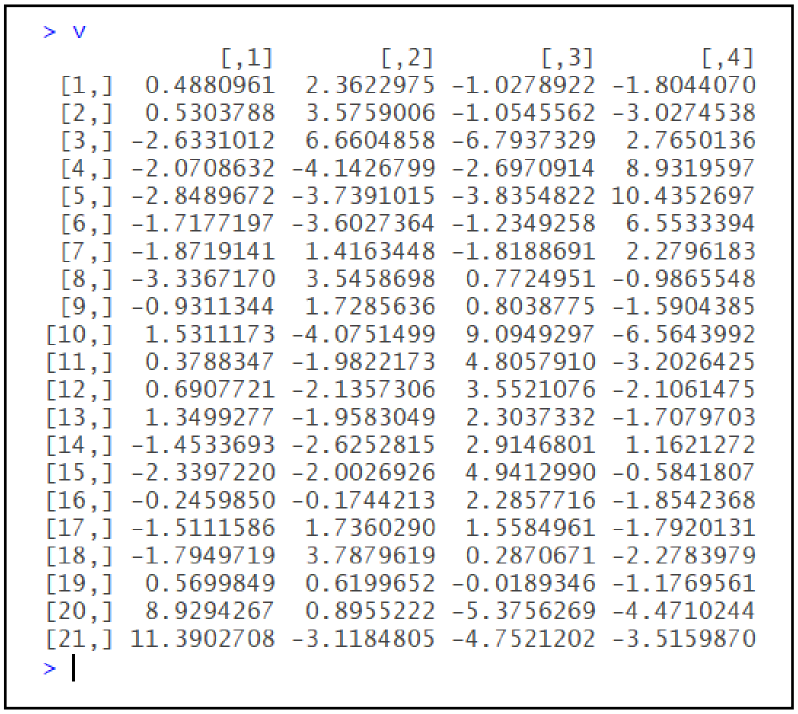
Above, you can see my implementation, whose details was discussed in the previous page. After each sample epoch, the final values of weights are used to calculate predicted classed and hence the objective value of the current iteration. If the change in the iteration decreases below a specific value, the algorithm would stop, which is not the case here since the process will be eventually limited by max_iterations before the desired convergence. On the right, you can see the development of the error function over iterations.



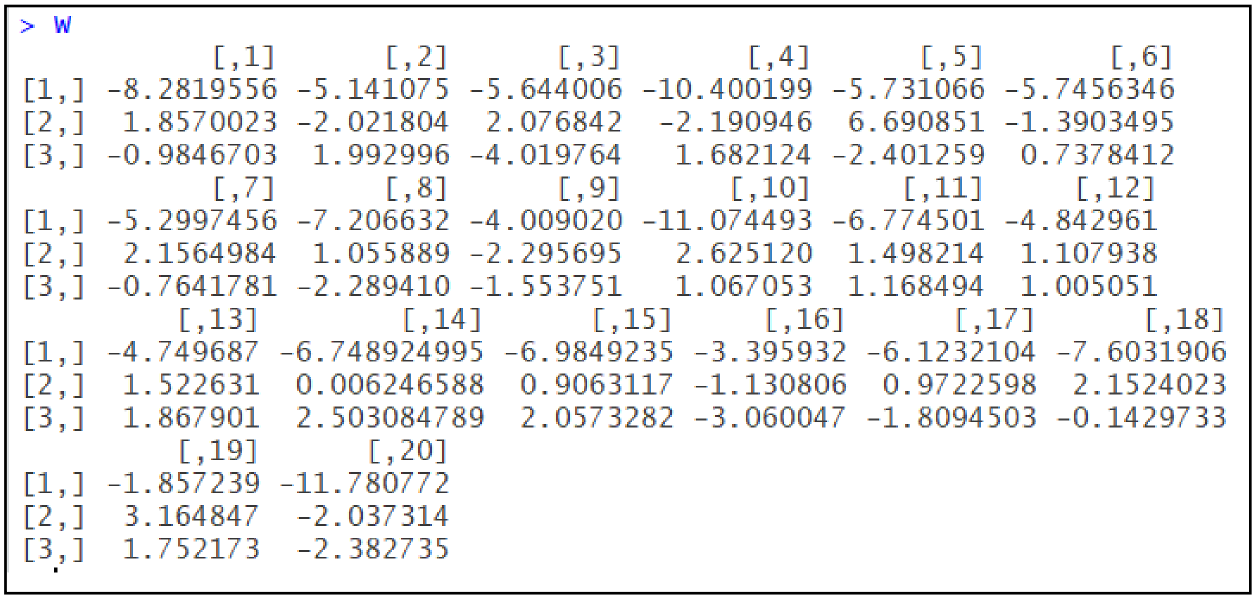




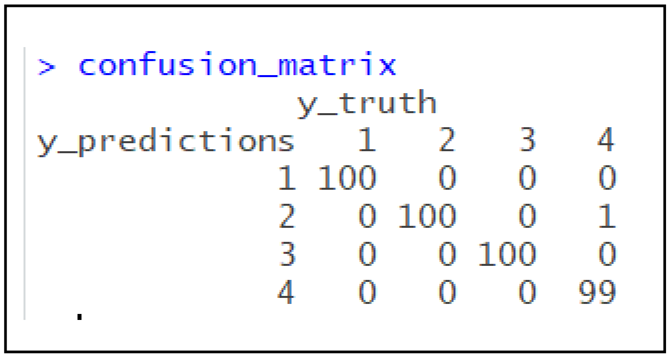
Here are two plots of decision regions, only with the difference of boundaries drawn. Again by chance, one can notice how symmetric and complementary the decision regions look like. Final values of weights W and v, and the confusion matrix is shown below.



```
> v
             [,1]       [,2]       [,3]       [,4]
 [1,]  0.4880961  2.3622975 -1.0278922 -1.8044070
 [2,]  0.5303788  3.5759006 -1.0545562 -3.0274538
 [3,] -2.6331012  6.6604858 -6.7937329  2.7650136
 [4,] -2.0708632 -4.1426799 -2.6970914  8.9319597
 [5,] -2.8489672 -3.7391015 -3.8354822 10.4352697
 [6,] -1.7177197 -3.6027364 -1.2349258  6.5533394
 [7,] -1.8719141  1.4163448 -1.8188691  2.2796183
 [8,] -3.3367170  3.5458698  0.7724951 -0.9865548
 [9,] -0.9311344  1.7285636  0.8038775 -1.5904385
[10,]  1.5311173 -4.0751499  9.0949297 -6.5643992
[11,]  0.3788347 -1.9822173  4.8057910 -3.2026425
[12,]  0.6907721 -2.1357306  3.5521076 -2.1061475
[13,]  1.3499277 -1.9583049  2.3037332 -1.7079703
[14,] -1.4533693 -2.6252815  2.9146801  1.1621272
[15,] -2.3397220 -2.0026926  4.9412990 -0.5841807
[16,] -0.2459850 -0.1744213  2.2857716 -1.8542368
[17,] -1.5111586  1.7360290  1.5584961 -1.7920131
[18,] -1.7949719  3.7879619  0.2870671 -2.2783979
[19,]  0.5699849  0.6199652 -0.0189346 -1.1769561
[20,]  8.9294267  0.8955222 -5.3756269 -4.4710244
[21,] 11.3902708 -3.1184805 -4.7521202 -3.5159870
> 
```



```
> W
          [,1]      [,2]      [,3]       [,4]      [,5]       [,6]
[1,] -8.2819556 -5.141075 -5.644006 -10.400199 -5.731066 -5.7456346
[2,]  1.8570023 -2.021804  2.076842  -2.190946  6.690851 -1.3903495
[3,] -0.9846703  1.992996 -4.019764   1.682124 -2.401259  0.7378412
          [,7]      [,8]      [,9]      [,10]     [,11]     [,12]
[1,] -5.2997456 -7.206632 -4.009020 -11.074493 -6.774501 -4.842961
[2,]  2.1564984  1.055889 -2.295695   2.625120  1.498214  1.107938
[3,] -0.7641781 -2.289410 -1.553751   1.067053  1.168494  1.005051
         [,13]        [,14]      [,15]     [,16]      [,17]      [,18]
[1,] -4.749687 -6.748924995 -6.9849235 -3.395932 -6.1232104 -7.6031906
[2,]  1.522631  0.006246588  0.9063117 -1.130806  0.9722598  2.1524023
[3,]  1.867901  2.503084789  2.0573282 -3.060047 -1.8094503 -0.1429733
         [,19]      [,20]
[1,] -1.857239 -11.780772
[2,]  3.164847  -2.037314
[3,]  1.752173  -2.382735
```



```
> confusion_matrix
              y_truth
y_predictions   1   2   3   4
            1 100   0   0   0
            2   0 100   0   1
            3   0   0 100   0
            4   0   0   0  99
```

Although the weight values are nearly impossible to interpret, we can see that they did not converge to zero or diverge. Only one point is classified incorrectly. This neural network with only one layer almost achieved the perfect result on the randomly fed training data.