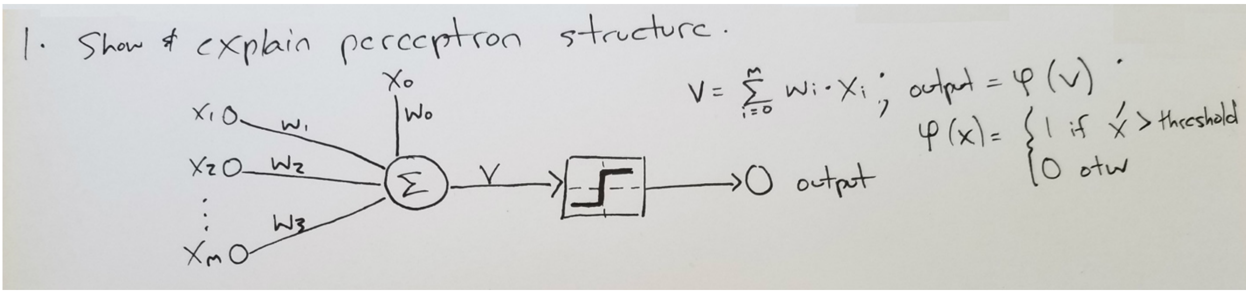
CAP5615 Justin Johnson
Summer 2018 Z23136514

Question 1

Show perceptron structure and explain function of each component:



The perceptron structure is drawn above. Input X is represented by a vector <x_1, x_2, … , x_m>. The vector is fed to the perceptron as inputs. An additional input, the bias, is also fed to the network as x_0. Bias is typically set to 1. Each input is multiplied by its corresponding weight, w_i, and then gathered in the summer function Σ. The summer function produces the weighted sum of each input, depicted as V in above image. The weighted sum V is then fed into an activation function ψ, which acts as a sign function that produces a binary output. The perceptron is either on or off, depending on the inputs, weights, and activation function.

What is the purpose of training examples in a neural network:
Training examples are used by the neural network to adjust its weights and fit to the data for the purpose of classification. The neural network updates weights when a training example is misclassified, minimizing error and learning the training data.

What is the expected output vs actual output of an example:
Expected output is the example's true label. Actual output is the output produced by the neural network. This output is produced by multiplying the inputs by their corresponding weights, calculating weighted sum, and feeding weighted sum through an activation function.

Question 2

What is a Perceptron Learning Rule:
Iterative process for updating perceptron input weights. Input weights are initialized as random, and then each misclassified training example is used to update the weights.

Explain Perceptron Learning Rule process:
The perceptron's input weights are initialized randomly in a pre-determined range, such as [-1, 1]. The Perceptron Learning Rule only updates weights when an instance is misclassified. The process gets the first misclassified example and calculates delta w for each input, that is, the

amount that the weight for each input should change. This is defined as the difference between desired output and actual output, multiplied by the learning rate and the value of the weight's corresponding input. The new weight for given instance is then updated to equal the previous weight plus the delta weight just calculated. This process is repeated until there are no more misclassified instances.

Question 3

What is the Gradient Descent Learning Rule:

Gradient Descent learning rule is another iterative process for updating a single layer network's weights during training. Unlike the Perceptron learning rule, which requires that the data set be linearly separable, the Gradient Descent learning rule will converge to the minimum error regardless of whether or not the data is linearly separable. Gradient Descent learning rule can be used to for classification or regression. The Gradient Descent learning rule relies on the networks squared error to update the weights. Since the squared error is a quadratic function, it has a global minimum, and its negative derivative can be used to take steps down hill toward the minimum. The Gradient Descent learning rule uses this negative derivative (gradient) to calculate delta weights. Unlike the Perceptron learning rule, the output does not pass through an activation function, and is therefore continuous.

Weight update rule:

$$w(k+1) = w(k) - \eta(gradient\ of\ E(W))$$

Before iterating over the training data, weights are initialized randomly. Then all training examples are processed to produce delta weights. For each pass over the training data, delta weights are initialized to zero, and then each instance is passed through the summation (weighted sum) function to produce an output. This output is subtracted from the desired output, and their difference is multiplied by the learning rate and the training instance to produce a delta weight. Delta weights are calculated for all training instances, then they are summed, and their total is used to update the weights. The weights are only updated after all training instances have been processed, and this process repeats until a minimum error is achieved or a max number of iterations is reached.

What is Delta Rule, what are the differences:
One downside to the Gradient Descent learning rule is that it becomes slow as the data set grows large, because the weights are only updated after the entire training set is iterated over. The Delta Rule provides a solution to this problem by randomly selecting 1 instance from the training set and using its output to update the weights. The update rule is the same as the previous Gradient Descent rule, the only difference is that the delta weight is calculated off the error of just one instance, instead of the entire training set. Since training instances are sampled randomly, the model can see a uniform representation of the training data in a much shorter amount of time.

Question 4

Prove that perceptron learning algorithm will terminate when given input s.t. input is linearly separable:

4. Given $C = C_1 \cup C_2$ s.t. $\exists$ hyperplane which separates C_1 and C_2.
Then Perceptron Rule applied to C will terminate after finite iterations K_{max}.

Proof Transform C_2 by replacing each x with $-x$,

$$\boxed{⓪ \text{ then } w(K+1) = \begin{cases} w(K) + \eta x(n) & \text{if } w^T(K) \cdot x(n) \le 0 \\ w(K) & \text{otw} \end{cases}}$$

Assume $\eta = 1$ and $w(1) = 0$.

Let $x(1) \dots x(K)$ be the sequence of inputs used over K iterations.

Then
$$\left.\begin{aligned} w(2) &= w(1) + x(1) \\ w(3) &= w(2) + x(2) \\ &\vdots \\ w(K) &= w(K-1) + x(K-1) \\ w(K+1) &= w(K) + x(K) \end{aligned}\right\}$$
All w's will cancel each other out, yielding $\Rightarrow$ $\boxed{w(K+1) = x(1) + \dots + x(K)}$ ①

Let w_* be the weight value that perfectly separates C_1 and C_2,
i.e. $\exists\, w_*$ s.t. $w_*^T x > 0 \quad \forall x \in C$

Let $\alpha = \min w_*^T x$ — a non-negative scalar which is minimum.

Then $w_*^T w(K+1) = w_*^T x(1) + \dots + w_*^T x(K) \ge K\alpha$
by multiplying both sides of ① by w_*^T. It must be greater than or equal to $K\alpha$ because α is the minimum value and their are K instances.

By Cauchy-Schwarz inequality $(\|v_1\|^2 \|v_2\|^2 \ge |v_1 \cdot v_2|^2)$,
$$\|w_*\|^2 \|w(K+1)\|^2 \ge \|w_*^T w(K+1)\|^2 = \|w_*^T x(1) + \dots + w_*^T x(K)\|^2 \ge (K\alpha)^2$$

$$\boxed{② \; \|w(K+1)\|^2 \ge \frac{K^2\alpha^2}{\|w_*\|^2}}$$
By dividing both sides by $\|w_*\|^2$

Next, Consider $w(K+1) = w(K) + x(K)$, by taking squared euclidean norm of both sides by expansion
$$\|w(K+1)\|^2 = \|w(K)\|^2 + \|x(K)\|^2 + 2w^T(K)x(K), \text{ per ⓪}$$
if misclassified, ~~then must be ≤ 0~~ then $2w^T(K)x(K) \le 0$,

Then we can convert to inequality: $\|w(K+1)\|^2 \le \|w(K)\|^2 + \|x(K)\|^2$

Yielding rules: $\|w(2)\|^2 \le \|w(1)\|^2 + \|x(1)\|^2$
$$\vdots$$
$$\|w(K+1)\|^2 \le \|w(K)\|^2 + \|x(K)\|^2$$
Again the weights will cancel out, and $\Rightarrow \boxed{\|w(K+1)\|^2 \le \sum_{i=1}^{K} \|x(i)\|^2}$ ③

Let $\beta = \max \|x(n)\|^2 \quad \forall x(n) \in C$

$\boxed{\text{Then } \|w(K+1)\|^2 \le K\beta}$ ④

Combining ② and ④ $\Rightarrow$ ⑤ $\boxed{\frac{K^2\alpha^2}{\|w_*\|^2} \le \|w(K+1)\|^2 \le K\beta}$

K^2 will increase faster than K as K grows larger. Therefore, K can not exceed K_{max}, to satisfy ⑤.

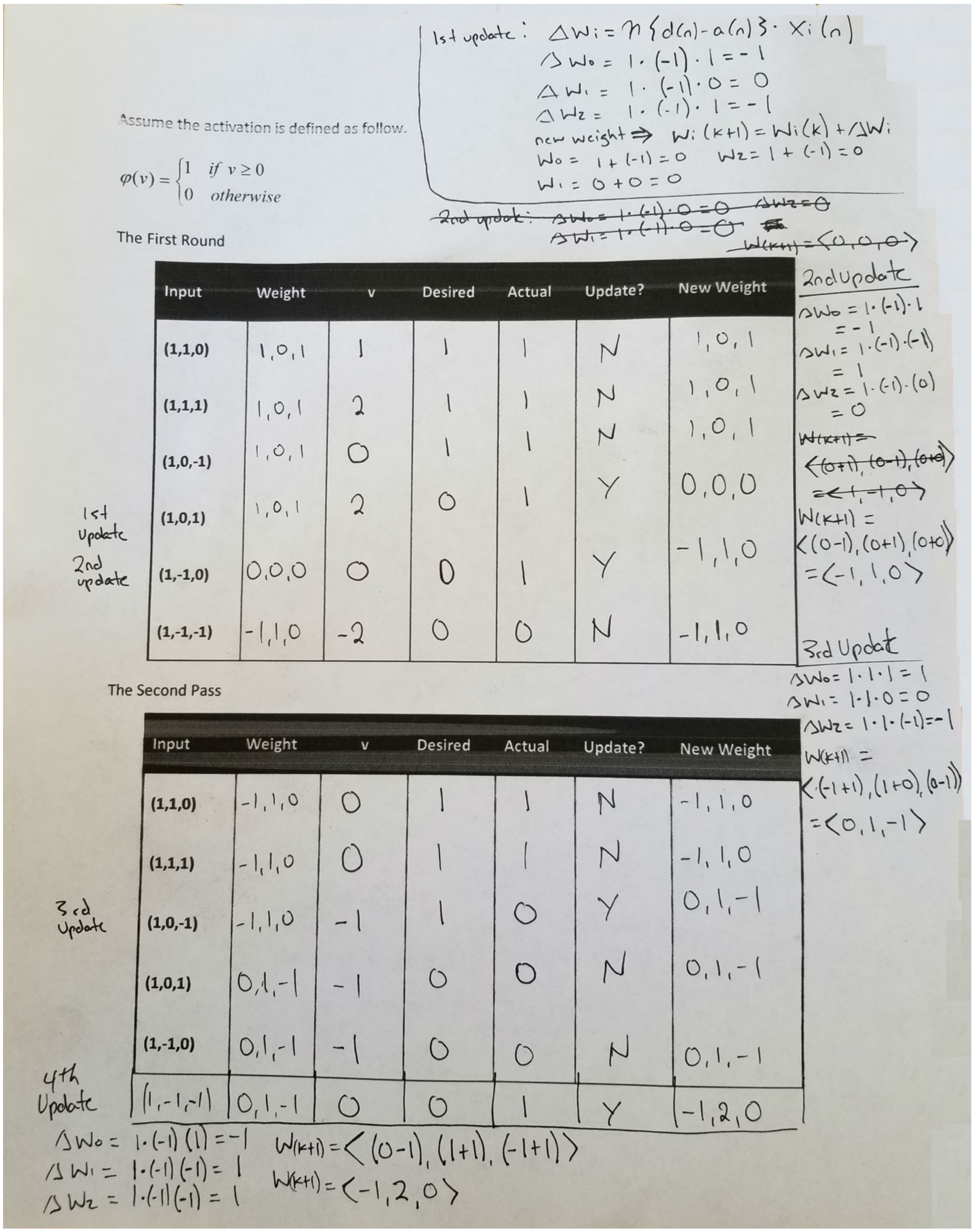
Solving for $K_{max} \Rightarrow \boxed{K_{max} = \frac{\|w_*\|^2}{\alpha^2}\beta}$ ⑥

Finally, K_{max} is defined by ⑥.

Question 5

Use perceptron learning rule to learn linear decision surface for two sets C1 and C2, given a learning rate of 1,and initial weights = <1, 0, 1>. List first two rounds using tables.

First two rounds and the weight change calculations are included in the below image:



1st update: $\Delta W_i = \eta \{d(n) - a(n)\} \cdot X_i(n)$

$\Delta W_0 = 1 \cdot (-1) \cdot 1 = -1$

$\Delta W_1 = 1 \cdot (-1) \cdot 0 = 0$

$\Delta W_2 = 1 \cdot (-1) \cdot 1 = -1$

new weight $\Rightarrow$ $W_i(k+1) = W_i(k) + \Delta W_i$

$W_0 = 1 + (-1) = 0$ $\quad W_2 = 1 + (-1) = 0$

$W_1 = 0 + 0 = 0$

~~2nd update: $\Delta W_0 = 1 \cdot (-1) \cdot 0 = 0$ $\Delta W_2 = 0$~~

~~$\Delta W_1 = 1 \cdot (-1) \cdot 0 = 0$~~

~~$W(k+1) = \langle 0, 0, 0 \rangle$~~

Assume the activation is defined as follow.

$$\varphi(v) = \begin{cases} 1 & \text{if } v \ge 0 \\ 0 & \text{otherwise} \end{cases}$$

The First Round

	Input	Weight	v	Desired	Actual	Update?	New Weight
	(1,1,0)	1,0,1	1	1	1	N	1,0,1
	(1,1,1)	1,0,1	2	1	1	N	1,0,1
	(1,0,-1)	1,0,1	0	1	1	N	1,0,1
1st Update	(1,0,1)	1,0,1	2	0	1	Y	0,0,0
2nd update	(1,-1,0)	0,0,0	0	0	1	Y	-1,1,0
	(1,-1,-1)	-1,1,0	-2	0	0	N	-1,1,0

2nd update

$\Delta W_0 = 1 \cdot (-1) \cdot 1 = -1$

$\Delta W_1 = 1 \cdot (-1) \cdot (-1) = 1$

$\Delta W_2 = 1 \cdot (-1) \cdot (0) = 0$

~~$W(k+1) = \langle (0+1), (0-1), (0+0) \rangle = \langle 1, -1, 0 \rangle$~~

$W(k+1) = \langle (0-1), (0+1), (0+0) \rangle = \langle -1, 1, 0 \rangle$

3rd Update

$\Delta W_0 = 1 \cdot 1 \cdot 1 = 1$

$\Delta W_1 = 1 \cdot 1 \cdot 0 = 0$

$\Delta W_2 = 1 \cdot 1 \cdot (-1) = -1$

$W(k+1) = \langle (-1+1), (1+0), (0-1) \rangle = \langle 0, 1, -1 \rangle$

The Second Pass

	Input	Weight	v	Desired	Actual	Update?	New Weight
	(1,1,0)	-1,1,0	0	1	1	N	-1,1,0
	(1,1,1)	-1,1,0	0	1	1	N	-1,1,0
3rd Update	(1,0,-1)	-1,1,0	-1	1	0	Y	0,1,-1
	(1,0,1)	0,1,-1	-1	0	0	N	0,1,-1
	(1,-1,0)	0,1,-1	-1	0	0	N	0,1,-1
4th Update	(1,-1,-1)	0,1,-1	0	0	1	Y	-1,2,0

$\Delta W_0 = 1 \cdot (-1)(1) = -1$ $\quad W(k+1) = \langle (0-1), (1+1), (-1+1) \rangle$

$\Delta W_1 = 1 \cdot (-1)(-1) = 1$ $\quad W(k+1) = \langle -1, 2, 0 \rangle$

$\Delta W_2 = 1 \cdot (-1)(-1) = 1$

Question 6

Prediction is 1 if probability of 1 is greater than or equal to 0.5, otherwise prediction is 0:

Index	Probability of 1	Prediction	True Label
1	0.8	1	1
2	0.2	0	0
3	0.4	0	1
4	0.55	1	1
5	0.45	0	1
6	0.9	1	1
7	0.3	0	0
8	0.4	0	0
9	0.56	1	1
10	0.92	1	1

Report Confusion Matrix

		Prediction	
		0	1
Actual	0	3	0
	1	2	5

Accuracy = (TP + TN) / (TP + TN + FP + FN) = 8 / 10 = 0.80
True Positive Rate = TP / (TP + FN) = 5 / (5 + 2) = 5 / 7 = 0.71
False Positive Rate = FP / (FP + TN) = 0 / 3 = 0.00

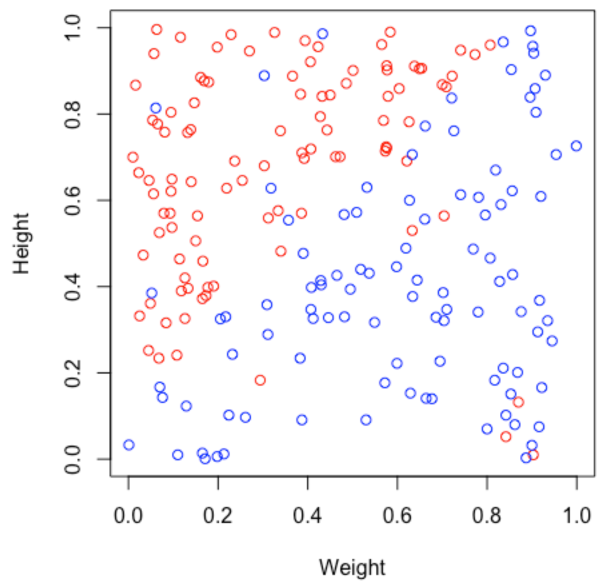
Question 7

Use Perceptron.R file to report results on Class1.txt and Class2.txt:

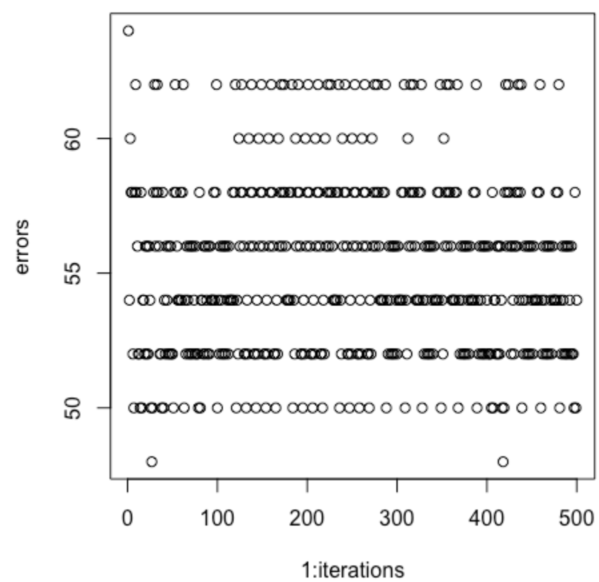
For this question, I pulled the perceptron function out of the Perceptron.R file. I did this so that I could write the R script to perform the tasks myself as a learning experience, instead of using the already defined code in Perceptron.R. You will see that on line 65 I load the perceptron file into memory, and then use its perceptron function to train weights against the randomized data. The source code for all tasks in Problem can be found on the next page.

```r
# define data
dataDir = 'data'
class1File = paste(dataDir, 'Class1.txt', sep = '/')
class2File = paste(dataDir, 'Class2.txt', sep = '/')

# load data
class1 = read.table(class1File, header = TRUE, sep = ',')
class2 = read.table(class2File, header = TRUE, sep = ',')

# add labels to data and update column names
# class 1 labelled with 1
class1Label = rep(1, nrow(class1))
class1 = cbind(class1, class1Label)
names(class1) = c('weight', 'height', 'label')
# class 2 labelled with 2
class2Label = rep(-1, nrow(class2))
class2 = cbind(class2, class2Label)
names(class2) = c('weight', 'height', 'label')


# PART 1
# PLOT DATA
# plot class1 red
plot(
  class1$weight,
  class1$height,
  xlim=c(0:1),
  ylim=c(0:1),
  xlab = 'Weight',
  ylab = 'Height',
  col='red'
)

# plot class2 in blue
points(
  class2$weight,
  class2$height,
  col = 'blue'
)

# PART 2
# PERCEPTRON LEARNER

# combine data
combinedData = rbind(class1, class2)

# add bias column to data
bias = rep(1, nrow(combinedData))
fullDataSet = cbind(bias, combinedData)

# randomize the data
ranIndices = sample(nrow(fullDataSet))
randomizedData = fullDataSet[ranIndices, ]

# preview data and confirm randomization worked
head(randomizedData)

# define hyperparameters
iterations = 500
learningRate = 0.05

# load perceptron function and run with randomized data
source('perceptron.R')
result = perceptron(randomizedData, learningRate, iterations)
trainedWeights = result$v1
errors = result$v2

# view the final weights
print(trainedWeights)

# plot iterations vs errors
plot(1:iterations, errors)

# plot decision boundary
plot(class1$weight, class1$height, xlim=c(0:1), ylim=c(0:1), col="red", xlab='weight', ylab='height')
points(class2$weight, class2$height, col="blue")
slope = trainedWeights[2] / trainedWeights[3]*(-1)
intercept = trainedWeights[1] / trainedWeights[3]*(-1)
abline(intercept, slope, col="green", lty=2)

# print decision boundary details
print(trainedWeights)
print(slope)
print(intercept)
```

Report scatter plot of all 200 instances:



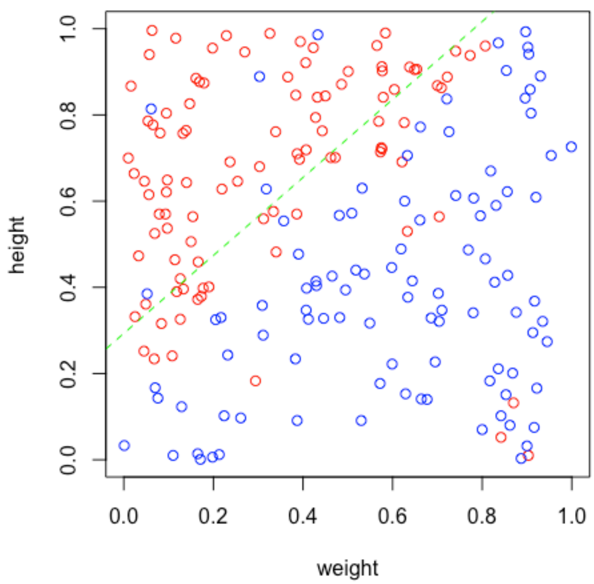
Use learning rate 0.05 and 500 iterations to train and report error rates of the perceptron learner:



Report the final weight values, slope, and y-intercept of decision surface:

```
> # print decision boundary details
> print(trainedWeights)
[1] -0.1000 -0.3085  0.3411
> print(slope)
[1] 0.9044269
> print(intercept)
[1] 0.2931692
```

Report decision surface on the original scatter plot:



Question 8

Gradient Descent learner was trained using Class1 and Class2. Below is screen shot of the learning algorithm:

```r
# Perceptron with Gradient Descent Learning Rule
perceptron = function(data, learnRate, errorThreshold, epochs) {
  # initialize weights
  weight <- getRandomWeights(dim(data)[2]-1)
  # initialize errors
  errors = rep(0, epochs)
  # extract features and labels from data
  label.index<-length(data[1,])
  features<-data[,-label.index]
  labels<-data[,label.index]

  # calculate initial system error
  systemError = getSystemError(features, weight, labels)
  print('systemerror')
  print(systemError)

  # while error exist and epochs not reached
  # loop over entire data set
  iter = 1
  while (systemError > errorThreshold & iter <= epochs) {

    # initialize deltaWeights to 0
    deltaWeight = rep(0, dim(data)[2]-1)
    squaredError = 0

    # iterate over all instances of data set
    # 1. calculate output
    # 2. calculate error d(n) - o(n)
    # 3. add error to this epochs running total error
    # 4. calculate weight difference for this instance
    # 5. add weight difference to epoch's running delta weight total
    for (ii in 1:nrow(data)) {
      ypred = sum(weight[1:length(weight)] * as.numeric(features[ii,]))
      err = labels[ii] - ypred
      squaredError = squaredError + (err * err)
      weightDiff = learnRate * (as.numeric(labels[ii]) - ypred) * as.numeric(features[ii,])
      deltaWeight = deltaWeight + weightDiff
    }
    # record errors
    errors[iter] = squaredError / nrow(data) / 2
    systemError = errors[iter]
    # record epoch's error
    systemError = errors[iter]
    # update system's weights
    weight = weight + (deltaWeight / nrow(data))
    iter = iter + 1
  }
  return(list(v1=weight,v2=errors))
}
```

Each data set was split randomly 80/20 and then their corresponding parts were combined to form training and test sets:

```r
# define data
dataDir = '../data'
class1File = paste(dataDir, 'Class1.txt', sep = '/')
class2File = paste(dataDir, 'Class2.txt', sep = '/')

# load data
class1 = read.table(class1File, header = TRUE, sep = ',')
class2 = read.table(class2File, header = TRUE, sep = ',')

# add labels to data and update column names
# class 1 labelled with 1
class1Label = rep(1, nrow(class1))
class1 = cbind(class1, class1Label)
names(class1) = c('weight', 'height', 'label')
# class 2 labelled with -1
class2Label = rep(-1, nrow(class2))
class2 = cbind(class2, class2Label)
names(class2) = c('weight', 'height', 'label')

# add bias columns to data
bias = rep(1, nrow(class1))
class1 = cbind(bias, class1)
class2 = cbind(bias, class2)

# shuffle data
class1 = class1[sample(nrow(class1)),]
class2 = class2[sample(nrow(class2)),]

# split both class1 and class2 80/20 (train/test)
class1.trainIndices = sample(nrow(class1) * 0.8)
class1TrainSet = class1[class1.trainIndices,]
class1TestSet = class1[-class1.trainIndices,]
class2.trainIndices = sample(nrow(class2) * 0.8)
class2TrainSet = class2[class2.trainIndices,]
class2TestSet = class2[-class2.trainIndices,]

# combine classes
combinedTrainSet = rbind(class1TrainSet, class2TrainSet)
combinedTestSet = rbind(class1TestSet, class2TestSet)


# PERFORM GRADIENT DESCENT

# define hyperparameters
iterations = 2000
learningRate = 0.05
errorThreshold = 0.1

# load gradient descent and run against training data
source('gradient-descent.R')
result = perceptron(combinedTrainSet, learningRate, errorThreshold, iterations)
trainedWeights = result$v1
errors = result$v2
historicWeights = result$v3

```

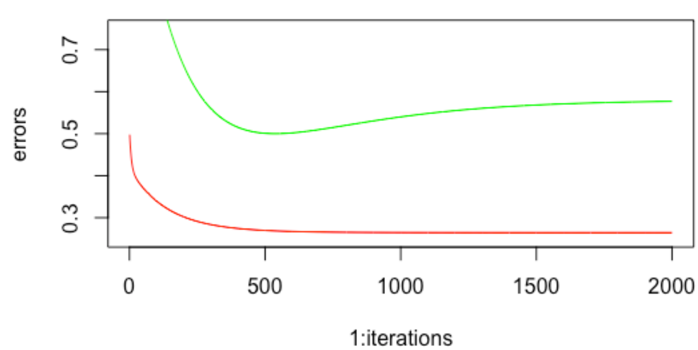
Then the trained model was fed the test set and the confusion matrix was calculated to determine accuracy:

```r

# use the weights to perform classification on test set
# extract features and labels from data
label.index<-length(combinedTestSet[1,])
testFeatures<-combinedTestSet[,-label.index]
testLabels<-combinedTestSet[,label.index]
testPredictions = rep(0, nrow(testFeatures))
for (ii in 1:nrow(combinedTestSet)) {
  v = sum(trainedWeights[1:length(trainedWeights)] * as.numeric(testFeatures[ii,]))
  if (v > 0) {
    testPredictions[ii] = 1
  } else {
    testPredictions[ii] = -1
  }
}

confMatrix = table(testPredictions, testLabels)
confMatrix

```

```
> confMatrix = table(testPredictions, testLabels)
> confMatrix
               testLabels
testPredictions -1  1
             -1 16  1
             1   4 19
```

Test Set Accuracy = 35 / 40 = 87.5%

The weights generated during the training process were stored in a matrix and the test set was evaluated against these weights for the purpose of viewing test set accuracy in comparison to training set accuracy. This procedure can help to detect overfitting. The below plot shows the training error in red and evaluation error in green. After 500 iterations the test set error starts to increase, due to the model fitting closer and closer to the training data, preventing it from generalizing to new data.



Finally, the training instances were plotted a long with the resulting decision boundary: