Mohammad Saleh
GRA at PHY WSU
m.saleh@cern.ch
fa9520@wayne.edu
April 14, 2017

CSC 5825
Homework 4

Solutions:

Question 1:

Comparison

In single-link, also known as the nearest neighbor technique or minimum method, in this technique the distance between clusters is defined as the distance between the closest pair, where only pairs consistung of one object from each cluster is considered. In complete-link, also known as farthest neighbor, it is opposite of single linkage, distance between clusters is defined as the distance between the most distant pair of object, one from each group. In average-linkage, the distance between two clusters is defined as the average distance between all pairs of object, where pair is made up of one object from each cluster.

Advantages, Disadvantages

The disdvantege of single-linkage is that sometimes it can produce chaining among the clusters where several clusters may be joined together simply because one of their cases is within close proximity of case from a separate cluster, this chaining effect can have disastrous effects on the cluster solution, also it is more sensitive to noise and outliners whcih can produce long, elongated clusters, where it advantages over the other is that it can handle non-elliptical shapes clusters. The complete-link advantages is that it is less susceptible to noise and produce more balanced clusters and it's disadvantages that it breaks large clusters and all of the clusters tends to have the same diameter, where small diameter clusters merged with large ones. For the average-linkage which compromise between Single and Complete-link, its main advantage that it is less susceptible to noise and outliers where its disadvantages is that it is biased towards globular clusters.

Question 2

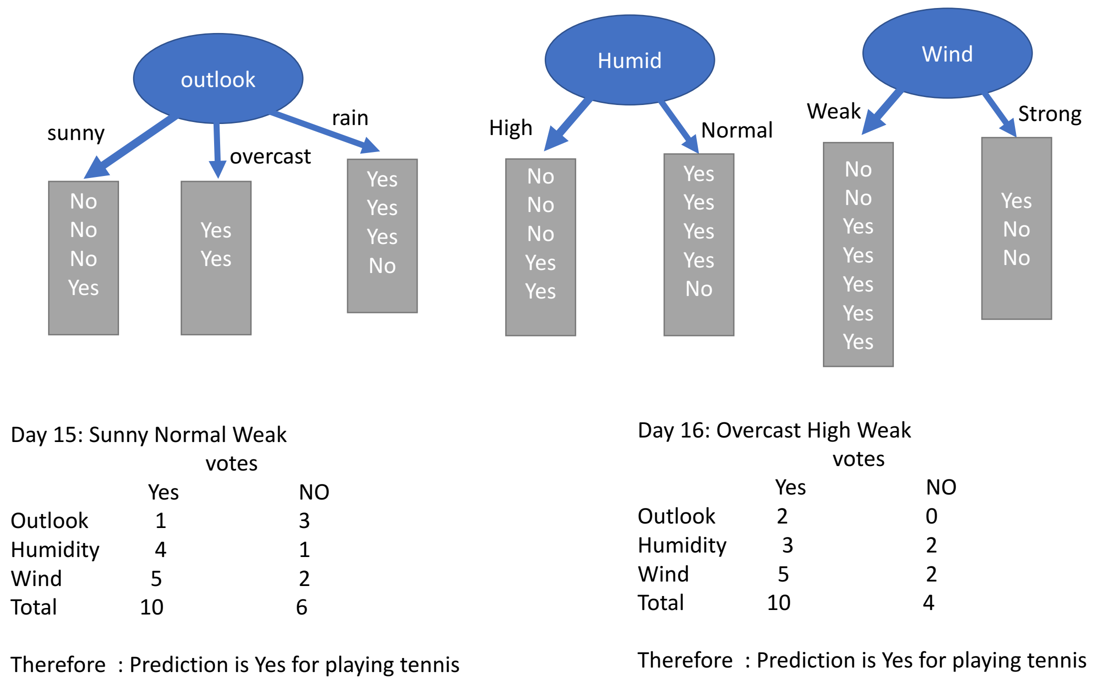
The solution for the two parts is in the figure 1.

Programming Questions

Question 1:

Below is the syntax for my code and figure.2 shows a snapshot of the compiled code the misclassification percentage error as we increase the number of iterations this error decrease.

```python
import numpy as np
#import pandas as pd
#MLP with Back-Propagation Neural
import math
import random
###### Initial matrix
def InitMatrix(I, J, f=0.0):
    m = []
    for i in range(I):
        m.append([f]*J)
    return m
```



Day 15: Sunny Normal Weak

votes

	Yes	NO
Outlook	1	3
Humidity	4	1
Wind	5	2
Total	10	6

Therefore : Prediction is Yes for playing tennis

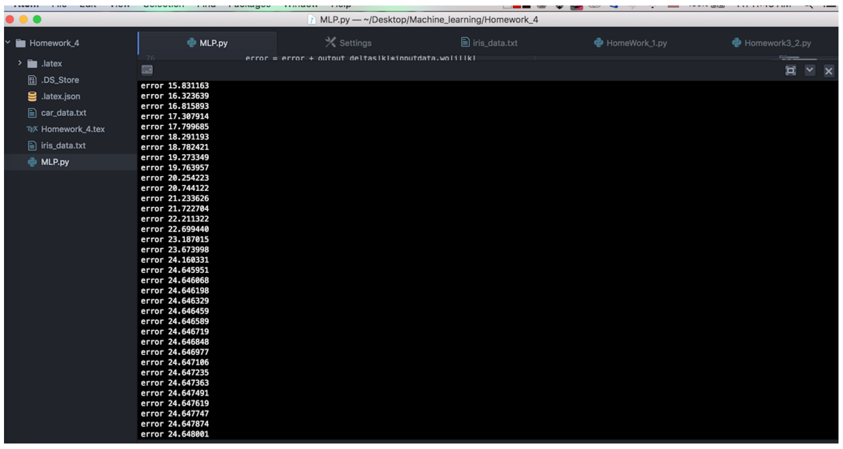
Day 16: Overcast High Weak

votes

	Yes	NO
Outlook	2	0
Humidity	3	2
Wind	5	2
Total	10	4

Therefore : Prediction is Yes for playing tennis

Figure 1: Tree stump.



```
error = error + output_deltas[k]*inputdata.wo[j][k]
```

```
error 15.831163
error 16.323639
error 16.815893
error 17.307914
error 17.799685
error 18.291193
error 18.782421
error 19.273349
error 19.763957
error 20.254223
error 20.744122
error 21.233626
error 21.722704
error 22.211322
error 22.699440
error 23.187015
error 23.673998
error 24.160331
error 24.645951
error 24.646068
error 24.646198
error 24.646329
error 24.646459
error 24.646589
error 24.646719
error 24.646848
error 24.646977
error 24.647106
error 24.647235
error 24.647363
error 24.647491
error 24.647619
error 24.647747
error 24.647874
error 24.648001
```

Figure 2: snapshot of my code after compiling.

```
# random initial weights
def rand(a, b):
   return (b-a)*random.random() + a

 #########or this sigmoid ###############
def sigmoid(x):
   return math.tanh(x)
 #########or this sigmoid ###############
#def sigmoid(x):
#    return (1/(1+math.exp(-1*x))
 #derivative of our sigmoid function the tan
def dsigmoid(y):
   return 1.0 - y**2

class MLP:             ## MLP class whcih take input features number and also how many hidden
    layers and the number of ouput nodes
   def __init__(inputdata, ni, hidden, no):
      inputdata.ni = ni + 1 # bais term
      inputdata.hidden = hidden
      inputdata.no = no
      inputdata.wi = InitMatrix(inputdata.ni, inputdata.hidden)
      inputdata.wo = InitMatrix(inputdata.hidden, inputdata.no)
      inputdata.ai = [1.0]*inputdata.ni
      inputdata.ah = [1.0]*inputdata.hidden
      inputdata.ao = [1.0]*inputdata.no

      for i in range(inputdata.ni):
         for j in range(inputdata.hidden):
            inputdata.wi[i][j] = rand(-0.2, 0.2)
      for j in range(inputdata.hidden):
         for k in range(inputdata.no):
            inputdata.wo[j][k] = rand(-2.0, 2.0)
      inputdata.ci = InitMatrix(inputdata.ni, inputdata.hidden)
      inputdata.co = InitMatrix(inputdata.hidden, inputdata.no)

   def iteratechange(inputdata, inputs):
      if len(inputs) != inputdata.ni-1:
         raise ValueError('wrong number of inputs')
      for i in range(inputdata.ni-1):
         inputdata.ai[i] = inputs[i]
      for j in range(inputdata.hidden):
         sum = 0.0
         for i in range(inputdata.ni):
            sum = sum + inputdata.ai[i] * inputdata.wi[i][j]
         inputdata.ah[j] = sigmoid(sum)

      for k in range(inputdata.no): #### output
         sum = 0.0
         for j in range(inputdata.hidden):
            sum = sum + inputdata.ah[j] * inputdata.wo[j][k]
         inputdata.ao[k] = sigmoid(sum)
      return inputdata.ao[:]


   def backPropagate(inputdata, targets, N, M):
      output_deltas = [0.0] * inputdata.no
      for k in range(inputdata.no):
         error = targets[k]-inputdata.ao[k]
         output_deltas[k] = dsigmoid(inputdata.ao[k]) * error
```

```
        H_diff = [0.0] * inputdata.hidden
        for j in range(inputdata.hidden):
            error = 0.0
            for k in range(inputdata.no):
                error = error + output_deltas[k]*inputdata.wo[j][k]
            H_diff[j] = dsigmoid(inputdata.ah[j]) * error

        for j in range(inputdata.hidden):
            for k in range(inputdata.no):
                change = output_deltas[k]*inputdata.ah[j]
                inputdata.wo[j][k] = inputdata.wo[j][k] + N*change + M*inputdata.co[j][k]
                inputdata.co[j][k] = change

        for i in range(inputdata.ni):
            for j in range(inputdata.hidden):
                change = H_diff[j]*inputdata.ai[i]
                inputdata.wi[i][j] = inputdata.wi[i][j] + N*change + M*inputdata.ci[i][j]
                inputdata.ci[i][j] = change

        error = 0.0
        for k in range(len(targets)):
        #    error = error + 0.5*(targets[k]-inputdata.ao[k])**2 #regres
             error = error - targets[k]*math.log(math.fabs(inputdata.ao[k]),10)

        return error ###### error#######

    def weights(inputdata):
        print('Input weights:')
        for i in range(inputdata.ni):
            print(inputdata.wi[i])
        print()
        print('Output weights:')
        for j in range(inputdata.hidden):
            print(inputdata.wo[j])

    def Iterate(inputdata, structure, iterations=10, N=0.005, M=0001): ### N for the Iterateing
         rate
        for i in range(iterations):
            error = 0.0
            for p in structure:
                inputs = p[0]
                targets = p[1]
                inputdata.iteratechange(inputs)
                error = error + inputdata.backPropagate(targets, N, M)
                print('error for that it %f' % error)

datainput =[] ### input data, it is minimized for clarity

Test= MLP(4, 1, 1)
Test.Iterate(datainput)
```

Bonus question

If we have direct weights from the input to the ouput and no activation function. It will be the same case for linear regression. So we end up looking at the advantages of using linear regression over the MLP. So it will be helpful to compute the optimal model directly and efficiently becuase if we add an activation function, and possibly one hidden layers, you cannot compute an optimal model directly anymore, and we are forced to use an iterative solution and no guarantee to converge and it will be slower. It can be trained as we usually do for linear regression and naive bayes in case of discrete target. Where as the disadvantages of not using the hidden layers is that linear reggression can only model linear function, where if we add hidden layers, we can approximate any continous function.