```
# Code from Chapter 4 of Machine Learning: An Algorithmic Perspective (2nd Edition)
# by Stephen Marsland (http://stephenmonika.net)

# You are free to use, change, or redistribute the code in any way you wish for
# non-commercial purposes, but please maintain the name of the original author.
# This code comes with no warranty of any kind.

# Stephen Marsland, 2008, 2014

import numpy as np

class mlp:
    """ A Multi-Layer Perceptron"""

    def __init__(self,inputs,targets,nhidden,beta=1,momentum=0.9,outtype='logistic'):
        """ Constructor """
        # Set up network size

        #funcion Shape devuelve una tupla de la estructura enviada
        #la indexacion devuelve la posicion de la tupla

        self.nin = np.shape(inputs)[1]
        self.nout = np.shape(targets)[1]
        self.ndata = np.shape(inputs)[0]
        self.nhidden = nhidden

        self.beta = beta
        self.momentum = momentum
        self.outtype = outtype

        # Initialise network
```

```
        self.weights1 = (np.random.rand(self.nin+1,self.nhidden)-0.5)*2/np.sqrt(self.nin)
        self.weights2 = (np.random.rand(self.nhidden+1,self.nout)-0.5)*2/np.sqrt(self.nhidden)

    def earlystopping(self,inputs,targets,valid,validtargets,eta,niterations=100):

        valid = np.concatenate((valid,-np.ones((np.shape(valid)[0],1))),axis=1)

        old_val_error1 = 100002
        old_val_error2 = 100001
        new_val_error = 100000

        count = 0
        while (((old_val_error1 - new_val_error) > 0.001) or ((old_val_error2 - old_val_error1)>0.001)):
            count+=1
            print count
            self.mlptrain(inputs,targets,eta,niterations)
            old_val_error2 = old_val_error1
            old_val_error1 = new_val_error
            validout = self.mlpfwd(valid)
            new_val_error = 0.5*np.sum((validtargets-validout)**2)

        print "Stopped", new_val_error,old_val_error1, old_val_error2
        return new_val_error

    def mlptrain(self,inputs,targets,eta,niterations):
        """ Train the thing """
        # Add the inputs that match the bias node
        #Pega a las entradas el valor -1 como el umbral
        inputs = np.concatenate((inputs,-np.ones((self.ndata,1))),axis=1)
        #Lista de 0 al numero de datos
        change = range(self.ndata)
```

```
#Prepara las matrices de pesos inicialmente en 0
updatew1 = np.zeros((np.shape(self.weights1)))
updatew2 = np.zeros((np.shape(self.weights2)))

for n in range(niterations):

    self.outputs = self.mlpfwd(inputs)

    error = 0.5*np.sum((self.outputs-targets)**2)
    if (np.mod(n,100)==0):
        print "Iteration: ",n, " Error: ",error

    # Different types of output neurons
    if self.outtype == 'linear':
      deltao = (self.outputs-targets)/self.ndata
    elif self.outtype == 'logistic':
      deltao = self.beta*(self.outputs-targets)*self.outputs*(1.0-self.outputs)
    elif self.outtype == 'softmax':
        deltao = (self.outputs-targets)*(self.outputs*(-self.outputs)+self.outputs)/self.ndata
    else:
      print "error"

    deltah = self.hidden*self.beta*(1.0-self.hidden)*(np.dot(deltao,np.transpose(self.weights2)))

    updatew1 = eta*(np.dot(np.transpose(inputs),deltah[:,:-1])) + self.momentum*updatew1
    updatew2 = eta*(np.dot(np.transpose(self.hidden),deltao)) + self.momentum*updatew2
    self.weights1 -= updatew1
    self.weights2 -= updatew2

    # Randomise order of inputs (not necessary for matrix-based calculation)
    #np.random.shuffle(change)
    #inputs = inputs[change,:]
```

```
        #targets = targets[change,:]

def mlpfwd(self,inputs):
    """ Run the network forward """

    # Multiplica la matriz de entradas que ya incluyen bias contra la matriz de pesos
    #en la cual tenia dudas porque anadia una fila mas (Finalmente para multiplicar
    #con el bias que se anadio)
    self.hidden = np.dot(inputs,self.weights1);
    #Se aplica funcion de activacion a la multiplicacion obtenida
    self.hidden = 1.0/(1.0+np.exp(-self.beta*self.hidden))
    #Vuelve y anade el bias al resultado obtenido
    self.hidden = np.concatenate((self.hidden,-np.ones((np.shape(inputs)[0],1))),axis=1)
    #Del cambio echo sobre la capa oculta multiplica contra los pesos de la siguiente conexion
    outputs = np.dot(self.hidden,self.weights2);

    # Different types of output neurons
    if self.outtype == 'linear':
     return outputs
    elif self.outtype == 'logistic':
        return 1.0/(1.0+np.exp(-self.beta*outputs))
    elif self.outtype == 'softmax':
        normalisers = np.sum(np.exp(outputs),axis=1)*np.ones((1,np.shape(outputs)[0]))
        return np.transpose(np.transpose(np.exp(outputs))/normalisers)
    else:
        print "error"

def confmat(self,inputs,targets):
    """Confusion matrix"""

    # Add the inputs that match the bias node
    inputs = np.concatenate((inputs,-np.ones((np.shape(inputs)[0],1))),axis=1)
```

```
outputs = self.mlpfwd(inputs)

nclasses = np.shape(targets)[1]

if nclasses==1:
    nclasses = 2
    outputs = np.where(outputs>0.5,1,0)
else:
    # 1-of-N encoding
    outputs = np.argmax(outputs,1)
    targets = np.argmax(targets,1)

cm = np.zeros((nclasses,nclasses))
for i in range(nclasses):
    for j in range(nclasses):
        cm[i,j] = np.sum(np.where(outputs==i,1,0)*np.where(targets==j,1,0))

print "Confusion matrix is:"
print cm
print "Percentage Correct: ",np.trace(cm)/np.sum(cm)*100
```