COMPUTATIONAL INTELLIGENCE - IT8601
ASSIGNMENT-3

To Implement the NEURO FUZZY Inference using python

Program:

init.py:

```
from anfis.membership import membershipfunction
from anfis.membership import mfDerivs
```

anfis.py:

```
# -*- coding: utf-8 -*-
"""
Created on Thu Apr 03 07:30:34 2014

@author: tim.meggs
"""
import itertools
import numpy as np
from membership import mfDerivs
import copy

class ANFIS:
    """Class to implement an Adaptive Network Fuzzy Inference System: ANFIS"

    Attributes:
        X
        Y
        XLen
        memClass
        memFuncs
        memFuncsByVariable
        rules
        consequents
        errors
        memFuncsHomo
        trainingType


    """

    def __init__(self, X, Y, memFunction):
        self.X = np.array(copy.copy(X))
        self.Y = np.array(copy.copy(Y))
        self.XLen = len(self.X)
        self.memClass = copy.deepcopy(memFunction)
```

```
        self.memFuncs = self.memClass.MFList
        self.memFuncsByVariable = [[x for x in range(len(self.memFuncs[z]))] for z in
range(len(self.memFuncs))]
        self.rules = np.array(list(itertools.product(*self.memFuncsByVariable)))
        self.consequents = np.empty(self.Y.ndim * len(self.rules) * (self.X.shape[1] + 1))
        self.consequents.fill(0)
        self.errors = np.empty(0)
        self.memFuncsHomo = all(len(i)==len(self.memFuncsByVariable[0]) for i in
self.memFuncsByVariable)
        self.trainingType = 'Not trained yet'

    def LSE(self, A, B, initialGamma = 1000.):
        coeffMat = A
        rhsMat = B
        S = np.eye(coeffMat.shape[1])*initialGamma
        x = np.zeros((coeffMat.shape[1],1)) # need to correct for multi-dim B
        for i in range(len(coeffMat[:,0])):
            a = coeffMat[i,:]
            b = np.array(rhsMat[i])
            S = S -
(np.array(np.dot(np.dot(np.dot(S,np.matrix(a).transpose()),np.matrix(a)),S)))/(1+(np.dot(np.dot(S,a),a)))
            x = x + (np.dot(S,np.dot(np.matrix(a).transpose(),(np.matrix(b)-np.dot(np.matrix(a),x)))))
        return x

    def trainHybridJangOffLine(self, epochs=5, tolerance=1e-5, initialGamma=1000, k=0.01):

        self.trainingType = 'trainHybridJangOffLine'
        convergence = False
        epoch = 1

        while (epoch < epochs) and (convergence is not True):

            #layer four: forward pass
            [layerFour, wSum, w] = forwardHalfPass(self, self.X)

            #layer five: least squares estimate
            layerFive = np.array(self.LSE(layerFour,self.Y,initialGamma))
            self.consequents = layerFive
            layerFive = np.dot(layerFour,layerFive)

            #error
            error = np.sum((self.Y-layerFive.T)**2)
            print('current error: '+ str(error))
            average_error = np.average(np.absolute(self.Y-layerFive.T))
```

```
        self.errors = np.append(self.errors,error)

        if len(self.errors) != 0:
            if self.errors[len(self.errors)-1] < tolerance:
                convergence = True

        # back propagation
        if convergence is not True:
            cols = range(len(self.X[0,:]))
            dE_dAlpha = list(backprop(self, colX, cols, wSum, w, layerFive) for colX in
range(self.X.shape[1]))


        if len(self.errors) >= 4:
            if (self.errors[-4] > self.errors[-3] > self.errors[-2] > self.errors[-1]):
                k = k * 1.1

        if len(self.errors) >= 5:
            if (self.errors[-1] < self.errors[-2]) and (self.errors[-3] < self.errors[-2]) and (self.errors[-3] <
self.errors[-4]) and (self.errors[-5] > self.errors[-4]):
                k = k * 0.9

        ## handling of variables with a different number of MFs
        t = []
        for x in range(len(dE_dAlpha)):
            for y in range(len(dE_dAlpha[x])):
                for z in range(len(dE_dAlpha[x][y])):
                    t.append(dE_dAlpha[x][y][z])

        eta = k / np.abs(np.sum(t))

        if(np.isinf(eta)):
            eta = k

        ## handling of variables with a different number of MFs
        dAlpha = copy.deepcopy(dE_dAlpha)
        if not(self.memFuncsHomo):
            for x in range(len(dE_dAlpha)):
                for y in range(len(dE_dAlpha[x])):
                    for z in range(len(dE_dAlpha[x][y])):
                        dAlpha[x][y][z] = -eta * dE_dAlpha[x][y][z]
        else:
            dAlpha = -eta * np.array(dE_dAlpha)
```

```
            for varsWithMemFuncs in range(len(self.memFuncs)):
                for MFs in range(len(self.memFuncsByVariable[varsWithMemFuncs])):
                    paramList = sorted(self.memFuncs[varsWithMemFuncs][MFs][1])
                    for param in range(len(paramList)):
                        self.memFuncs[varsWithMemFuncs][MFs][1][paramList[param]] =
self.memFuncs[varsWithMemFuncs][MFs][1][paramList[param]] +
dAlpha[varsWithMemFuncs][MFs][param]
            epoch = epoch + 1


        self.fittedValues = predict(self,self.X)
        self.residuals = self.Y - self.fittedValues[:,0]

        return self.fittedValues


    def plotErrors(self):
        if self.trainingType == 'Not trained yet':
            print(self.trainingType)
        else:
            import matplotlib.pyplot as plt
            plt.plot(range(len(self.errors)),self.errors,'ro', label='errors')
            plt.ylabel('error')
            plt.xlabel('epoch')
            plt.show()

    def plotMF(self, x, inputVar):
        import matplotlib.pyplot as plt
        from skfuzzy import gaussmf, gbellmf, sigmf

        for mf in range(len(self.memFuncs[inputVar])):
            if self.memFuncs[inputVar][mf][0] == 'gaussmf':
                y = gaussmf(x,**self.memClass.MFList[inputVar][mf][1])
            elif self.memFuncs[inputVar][mf][0] == 'gbellmf':
                y = gbellmf(x,**self.memClass.MFList[inputVar][mf][1])
            elif self.memFuncs[inputVar][mf][0] == 'sigmf':
                y = sigmf(x,**self.memClass.MFList[inputVar][mf][1])

            plt.plot(x,y,'r')

        plt.show()

    def plotResults(self):
```

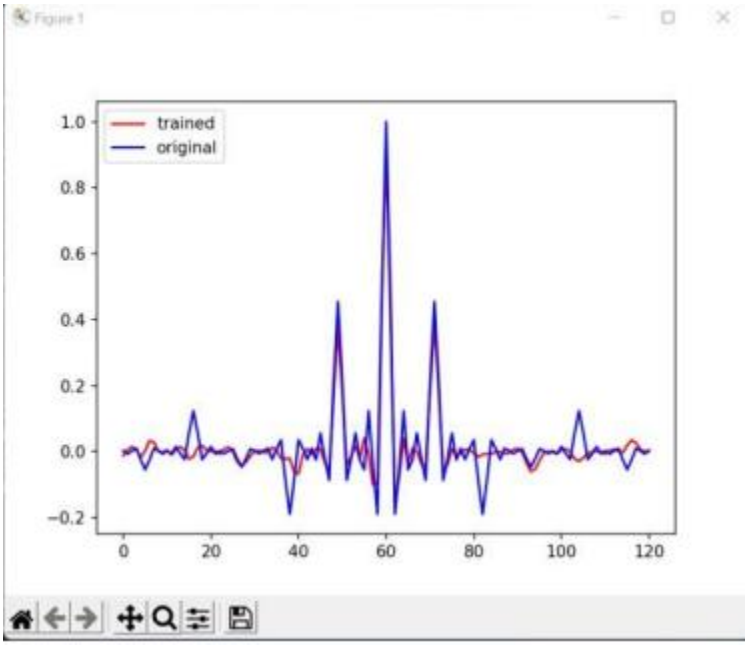
```
        if self.trainingType == 'Not trained yet':
            print(self.trainingType)
        else:
            import matplotlib.pyplot as plt
            plt.plot(range(len(self.fittedValues)),self.fittedValues,'r', label='trained')
            plt.plot(range(len(self.Y)),self.Y,'b', label='original')
            plt.legend(loc='upper left')
            plt.show()



def forwardHalfPass(ANFISObj, Xs):
    layerFour = np.empty(0,)
    wSum = []

    for pattern in range(len(Xs[:,0])):
        #layer one
        layerOne = ANFISObj.memClass.evaluateMF(Xs[pattern,:])

        #layer two
        miAlloc = [[layerOne[x][ANFISObj.rules[row][x]] for x in range(len(ANFISObj.rules[0]))] for row
in range(len(ANFISObj.rules))]
        layerTwo = np.array([np.product(x) for x in miAlloc]).T
        if pattern == 0:
            w = layerTwo
        else:
            w = np.vstack((w,layerTwo))

        #layer three
        wSum.append(np.sum(layerTwo))
        if pattern == 0:
            wNormalized = layerTwo/wSum[pattern]
        else:
            wNormalized = np.vstack((wNormalized,layerTwo/wSum[pattern]))

        #prep for layer four (bit of a hack)
        layerThree = layerTwo/wSum[pattern]
        rowHolder = np.concatenate([x*np.append(Xs[pattern,:],1) for x in layerThree])
        layerFour = np.append(layerFour,rowHolder)

    w = w.T
    wNormalized = wNormalized.T

    layerFour = np.array(np.array_split(layerFour,pattern + 1))
```

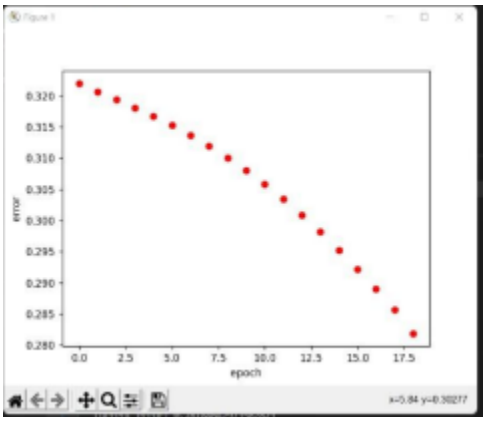
```
    return layerFour, wSum, w


def backprop(ANFISObj, columnX, columns, theWSum, theW, theLayerFive):

    paramGrp = [0]* len(ANFISObj.memFuncs[columnX])
    for MF in range(len(ANFISObj.memFuncs[columnX])):

        parameters = np.empty(len(ANFISObj.memFuncs[columnX][MF][1]))
        timesThru = 0
        for alpha in sorted(ANFISObj.memFuncs[columnX][MF][1].keys()):

            bucket3 = np.empty(len(ANFISObj.X))
            for rowX in range(len(ANFISObj.X)):
                varToTest = ANFISObj.X[rowX,columnX]
                tmpRow = np.empty(len(ANFISObj.memFuncs))
                tmpRow.fill(varToTest)

                bucket2 = np.empty(ANFISObj.Y.ndim)
                for colY in range(ANFISObj.Y.ndim):

                    rulesWithAlpha = np.array(np.where(ANFISObj.rules[:,columnX]==MF))[0]
                    adjCols = np.delete(columns,columnX)

                    senSit =
mfDerivs.partial_dMF(ANFISObj.X[rowX,columnX],ANFISObj.memFuncs[columnX][MF],alpha)
                    # produces d_ruleOutput/d_parameterWithinMF
                    dW_dAplha = senSit *
np.array([np.prod([ANFISObj.memClass.evaluateMF(tmpRow)[c][ANFISObj.rules[r][c]] for c in
adjCols]) for r in rulesWithAlpha])

                    bucket1 = np.empty(len(ANFISObj.rules[:,0]))
                    for consequent in range(len(ANFISObj.rules[:,0])):
                        fConsequent =
np.dot(np.append(ANFISObj.X[rowX,:],1.),ANFISObj.consequents[((ANFISObj.X.shape[1] + 1) *
consequent):(((ANFISObj.X.shape[1] + 1) * consequent) + (ANFISObj.X.shape[1] + 1)),colY])
                        acum = 0
                        if consequent in rulesWithAlpha:
                            acum = dW_dAplha[np.where(rulesWithAlpha==consequent)] * theWSum[rowX]

                        acum = acum - theW[consequent,rowX] * np.sum(dW_dAplha)
                        acum = acum / theWSum[rowX]**2
                        bucket1[consequent] = fConsequent * acum
```

```
                sum1 = np.sum(bucket1)

                if ANFISObj.Y.ndim == 1:
                    bucket2[colY] = sum1 * (ANFISObj.Y[rowX]-theLayerFive[rowX,colY])*(-2)
                else:
                    bucket2[colY] = sum1 * (ANFISObj.Y[rowX,colY]-theLayerFive[rowX,colY])*(-2)

            sum2 = np.sum(bucket2)
            bucket3[rowX] = sum2

        sum3 = np.sum(bucket3)
        parameters[timesThru] = sum3
        timesThru = timesThru + 1

    paramGrp[MF] = parameters

  return paramGrp


def predict(ANFISObj, varsToTest):

  [layerFour, wSum, w] = forwardHalfPass(ANFISObj, varsToTest)

  #layer five
  layerFive = np.dot(layerFour,ANFISObj.consequents)

  return layerFive


if __name__ == "__main__":
  print("I am main!")
tests.py:
import sys
import anfis
import numpy
import membership.mfDerivs
import membership.membershipfunction


# numpy.loadtxt('c:\\Python_fiddling\\myProject\\MF\\trainingSet.txt',usecols=[1,2,3])
ts = numpy.loadtxt("trainingSet.txt", usecols=[1, 2, 3])
X = ts[:, 0:2]
Y = ts[:, 2]
```

```
mf = [[['gaussmf', {'mean': 0., 'sigma': 1.}], ['gaussmf', {'mean': -1., 'sigma': 2.}], ['gaussmf', {'mean': -4.,
'sigma': 10.}], ['gaussmf', {'mean': -7., 'sigma': 7.}]],
     [['gaussmf', {'mean': 1., 'sigma': 2.}], ['gaussmf', {'mean': 2., 'sigma': 3.}], ['gaussmf', {'mean': -2.,
'sigma': 10.}], ['gaussmf', {'mean': -10.5, 'sigma': 5.}]]]


mfc = membership.membershipfunction.MemFuncs(mf)
anf = anfis.ANFIS(X, Y, mfc)
anf.trainHybridJangOffLine(epochs=20)
print(round(anf.consequents[-1][0], 6))
print(round(anf.consequents[-2][0], 6))
print(round(anf.fittedValues[9][0], 6))
if round(anf.consequents[-1][0], 6) == -5.275538 and round(anf.consequents[-2][0], 6) == -1.990703 and
round(anf.fittedValues[9][0], 6) == 0.002249:
    print('test is good')

print("Plotting errors")
anf.plotErrors()
print("Plotting results")
anf.plotResults()
```

Output:




0.320
0.315
0.310
0.305
0.300
0.295
0.290
0.285
0.280
error
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
epoch