Assignment 4a - Using Genetic Algorithms and Genetic Programs as Classifiers on Real-World Data

Joel Doumit
College of Computer Science
University of Idaho
Moscow, ID 83843
Email: doum6708@vandals.uidaho.edu

1. Output

```
GP_POP:
(sqrt(sqrt(sqrt(((((sqrt(((x27*x26)+(-3+1))+
((sqrtx22)-(x2/x16))))+((((sqrtx25)*(-1/x16))*
((x12-x26)+(x4+x13)))/(((x19+5)/(x7*3))/((sqrtx9)
+(x26+-2)))))+(((((x27+x20)/(sqrt4))/((x12+x18)*
(sqrtx12)))-(sqrt((x7*x4)+(x23+x21))))*
(sqrt(sqrt((x6--4)-(-1+1))))))*((sqrt(sqrt(((x10+x19)
/(sqrt4))-((5*-4)*(x4*x15)))))+(((((0*x14)+(-4*x0))*
((x13+x14)/(sqrtx3)))*(sqrt((2--5)/(sqrtx4))))*
((sqrt((x2+10)-(4--5)))-(sqrt((x22+-3)*(0-x23)))))))+
((((((x25+-5)*(x27-x11))+(sqrt(sqrtx13)))/(((1*10)-
(x24+x15))*(sqrt(x4/x8))))/((((-5-1)*(-1-x15))*
(sqrt(1/x29)))*(((x6*0)-(sqrt10))+((x18*x8)+
(x6*x24)))))/(((((sqrtx24)*(x20+x16))*((0-3)+
(10/x16)))*(((x10/-4)-(x28*x17))*(sqrt(x29--5))))
+(((sqrt(sqrtx16))-((-4-x7)+(2*-10)))+
(sqrt(sqrt(0-3))))))/(sqrt(sqrt((sqrt((x11-5)
/(x28+x25)))/(((x16/x16)/(x8+x7))-((x3*x29)-
(sqrtx28)))))))))))
Eval'd:
9.131085919036861

NNpop[0]:  [[-6.94753916e-02  2.16312456e-01  4.26660566e-05 ...  2.57374553e-13
   -2.27304669e-01 -1.92480507e-01]
 [-2.17214089e-01  5.85156040e-02 -1.91861002e-06 ...  4.06051029e-03
   1.65987551e-01  1.18591921e-01]
 [ 4.94268847e-02 -1.59537879e-01 -1.40667541e-03 ... -3.42656259e-04
   3.29395418e-02 -1.98770529e-01]
 ...
 [ 1.07438940e-01  1.43391525e-01 -2.81537724e-03 ... -1.06782519e-10
   2.69749232e-01 -7.13234983e-02]
 [ 1.52390497e-01  1.78463561e-01  7.60354470e-03 ... -1.93717393e-05
   -3.57852821e-02 -2.03433240e-01]
 [-1.46465835e-01  1.19024766e-01 -1.37559472e-02 ... -2.85114542e-06
   -9.56473858e-02  1.52835446e-01]]
Proof of Data Set Import:
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
```

```
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
['malignant' 'benign']
```

2. Code

```
#!/Library/Frameworks/Python.framework/Versions/3.4/bin/python3
# Joel Doumit
# CS472 - Evolutionary Algorithms
# Assignment 4a
# Tree code re-used from Assignment 3.

import sklearn
from sklearn.datasets import load_breast_cancer
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split

import numpy as np
import random
import csv

operators = ["+", "-", "*", "/", "sqrt"]
terminals = ["x0", "x1", "x2", "x3", "x4", "x5", "x6", "x7", "x8", "x9", \
"x10", "x11", "x12", "x13", "x14", "x15", "x16", "x17", "x18", "x19", "x20", \
"x21", "x22", "x23", "x24", "x25", "x26", "x27", "x28", "x29", \
-10, -5, -4, -3, -2, 0, -1, 1, 2, 3, 4, 5, 10]

class node:
    def __init__(self, value):
        self.left = None
        self.data = value
        self.right = None

def evaluateTree(root):
    # empty tree
    if root is None:
        return 0

    if isinstance(root.data, str) and root.data.startswith("x"):
        index = int(root.data[1:])
        return mlp.coefs_[0][0][index]

    # leaf node
    if root.left is None and root.right is None:
        return int(root.data)

    # evaluate left tree
    left_sum = evaluateTree(root.left)

    # evaluate right tree
    right_sum = evaluateTree(root.right)
```

```
    if root.data == '+':
        return left_sum + right_sum

    elif root.data == '-':
        return left_sum - right_sum

    elif root.data == '*':
        return left_sum * right_sum

    elif root.data == 'sqrt':
        if right_sum >= 0:
            return np.sqrt(right_sum)
        else:
            return np.sqrt(-1*right_sum)

    else:
        if(right_sum == 0):
            return 0
        else:
            return left_sum / right_sum

def printTree(root):
    if root is None:
        print("Nothing")
    else:
        if root.left is None and root.right is None:
            print(root.data, end='')
        else:
            if root.data == 'sqrt':
                print("(", end='')
                print(root.data, end='')
                printTree(root.right)
                print(")", end='')
            else:
                print("(", end='')
                printTree(root.left)
                print(root.data, end='')
                printTree(root.right)
                print(")", end='')

def nodeVal(operChanceLow, operChanceHigh):
    if (np.random.choice([0,1], p=[operChanceLow, operChanceHigh])):
        return random.choice(operators)
    else:
        return random.choice(terminals)


def generateTerminal():
    return random.choice(terminals)

def generateFull(depth=1):
    if depth < 12:
        root = node(nodeVal(0, 1))
        root.left = generateFull(depth+1)
        root.right = generateFull(depth+1)
        return root
    else:
        root = node(nodeVal(1,0))
```

```
        return root

def generateGrow(depth=1):
    if depth < 12:
        root = node(nodeVal(0.1, 0.9))
        if root.data in operators:
            root.left = generateGrow(depth+1)
            root.right = generateGrow(depth+1)
        return root
    else:
        root = node(nodeVal(1,0))
        return root

def Copy(source):
    if(source):
        new = node(source.data)
        new.data = source.data
        if source.left:
            new.left = (Copy(source.left))
        if source.right:
            new.right = (Copy(source.right))
    return new

def Delete(population, index):
    del population[index]

def keyFunction(x):
    return(-0.11*(x**5)+(x**3)+8*x)

def fitness(tree, keyAnswers, usePM, population):
    error = []
    answerArray = []
    sumErr = 0

    for i in range(5):
        answerArray.append(evaluateTree(tree, i+1))

    for i in range(5):
        error.append(answerArray[i] - keyAnswers[i])

    for i in error:
        sumErr += (i**2)
        mean = sumErr/len(error)
        RMSE = np.sqrt(mean)
        if usePM:
            if (maxDepth(tree)>6):
                parsimony = avgFit(population)
                fitness = RMSE + parsimony
                return fitness
        else:
            return RMSE

def avgFit(population):
    average = []
    for i in range(100):
        average.append(population[i][1])
    avg = sum(average)/len(average)
    return avg
```

```
def maxDepth(node):
    if node is None:
        return 0 ;
    else :
        # Compute the depth of each subtree
        lDepth = maxDepth(node.left)
        rDepth = maxDepth(node.right)
        # Use the larger one
        if (lDepth > rDepth):
            return lDepth+1
        else:
            return rDepth+1

def Offspring(parent1, parent2):
    child1 = Copy(parent1)
    child2 = Copy(parent2)
    n1 = np.random.choice(Nodelist(child1))
    n2 = np.random.choice(Nodelist(child2))
    n3 = node(None)
    n3.data = n1.data
    n3.left = n1.left
    n3.right = n1.right
    n1.data = n2.data
    n1.left = n2.left
    n1.right = n2.right
    n2.data = n3.data
    n2.left = n3.left
    n2.right = n3.right

    mutation(child1)
    mutation(child2)
    return child1, child2

def Nodelist(root):
    if(root):
        list = []
        list.append(root)
        if((root.left) and (root.right)):
            for l in Nodelist(root.left):
                list.append(l)
            for l in Nodelist(root.right):
                list.append(l)
        if(root.left):
            for l in Nodelist(root.left):
                list.append(l)
        if(root.right):
            for l in Nodelist(root.right):
                list.append(l)
    return list

def selectionPool(population):
    chosenIndiv = np.random.choice(100, 5, replace=False)
    selectPool = []
    for i in chosenIndiv:
        selectPool.append(population[i])
    return selectPool
```

```
def sortPool(to_sort):
    to_sort.sort(key=lambda x: x[1])
    return to_sort

def pickParents(sorted_pool):
    return sorted_pool[0][0], sorted_pool[1][0]

def mutation(kid):
    n1 = np.random.choice(Nodelist(kid))
    if n1.data in operators:
        n1.data = random.choice(operators)
    if n1.data in terminals:
        n1.data = random.choice(terminals)


def survivors(sortedPop):
    sortedPop = sortedPop[:len(sortedPop)-2]
    return sortedPop

def stats(sortedPop):
    best = sortedPop[0][1]
    avg = avgFit(sortedPop)
    return best, avg

#################################################################
############### MAIN ############################################
#################################################################

if __name__=='__main__':
    with open('Assign4_JSD.csv', mode='w', newline = '') as outfile:
        writer = csv.writer(outfile)
        writer.writerow(['Generation','Best','Avg'])

        cancer = load_breast_cancer()

        x_train, x_test, y_train, y_test = train_test_split(cancer.data, cancer.target)

        mlp = MLPClassifier(random_state=42)
        mlp.fit(x_train, y_train)

        # bestTrees =[]
        # for k in range(15):
        #   keyAnswers =[]
        #   for i in range(5):
        #       keyAnswers.append(keyFunction(i+1))

        # Initialization
        Population = []
        for i in range(50):
            tree = generateFull()
            Population.append(tree)

        for i in range(50):
            tree = generateGrow()
            Population.append(tree)

        # Selection
        #   for g in range(500):
```