Assignment 2: Symbolic Regression

Submitted by: Yaoyao Xu (UNI: yx2725)

Ruoyao Zhao (UNI: rz2535)

Course number: MECSE4510

Instructor: Hod Lipston

Submitted time: 10/24/2021

Grace Hour Used: 1 h

Grace Hour Left: 105 h

**Results**

**Chart, line chart

Description automatically generated**

Figure 1: Function Generated with Symbolic Regression

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Evaluation** | **MEA** | **Function** |
| EA + 30% + Tournament | 10000 | **0.09** |  |

**Function Expressed in Tree Structure**

**Text

Description automatically generated**

**Representation**

The binary heap data structure was used to represent the function expression. The root position must be an operator, and the children of element i could be an operator, a variable, or a constant. When the children consist of the operator, there should be sub-tree under it, and the depth of the heap will increase.

**Method**

**Random Search**

For random search, a function is generated randomly using operators including “sin, cos, +, -, \*, /”, constants, and variables in each evaluation. The fitness is computed by replacing the variables with each x values of the real data and evaluate all the operators.

**Hill Climber**

For Hill Climber, we first generate the heaps randomly in one population size and make it the first generation. Then the mutation is applied to create the next generation. For constant, we will change it by a small amount or insert a small factor. After the mutation, we do the selection among the current population (current generation plus the next generation), then we repeat this whole process until the last evaluation.

**Genetic Programming**

Compared with the Hill Climber, GP consists of crossover in addition to mutation, the cross over will randomly change the sub-trees which begin with the cell includes the operators of two parents, and then we check if these new heaps are correct or not. Then all the current population will be evaluated to get the fitness. After this, we do the selection again among the current population and repeat the whole process until the last evaluation.

* EA + random selection (represented by EA):

The initial population is set to be 1000. In each evolution, two parents were randomly selected to perform crossover and mutation to produce two children. Two children are compared with their parents. If the child’s error is lower than the parent, the parent is substituted by the child. The lowest error is recorded each time.

* EA + rank selection + 50% selection pressure (represented by EA 50%):

The first election pressure is 50%. Instead of a steady-state selection, we initial a 500 population and select two children in the top 50% of the population, forming 1000 candidates. 1000 candidates are sorted based on their error, and the top 50 percent survive for the next evaluation.

* EA + rank selection + 30% selection pressure (represented by EA 30%):

The second variation is to use selection pressure of 30%. The process is similar to the 50% selection process.

**Selection**

Tournament method was used to select the best fitness individuals. Two parents were randomly selected and compared by their fitness, the better one will be chosen.

**Performance and Correctness**

The error is normalized by sqrt (4) because each algorithm will be run for four times. The validation set is recorded every 1000 evolutions (10 points total and only EA is plotted with validation). The validation plots max(training[i], validation[i]) for each point. The y axis shows the mean error average (MEA).

**Simple Test Example**

The figure below shows the testing example. 50 points are produced with equation sin(x) + cos(x) + 1. The EA + 30% selection pressure was applied to the dataset (blue points). There are only a few points that deviate from the predicted equation.

Chart

Description automatically generated with medium confidence

Figure 2: Simple Test

**Performance Analysis**

The random search and hill climber acts as two baselines for the optimization method. The EA + steady state with random selection performs slightly better than hill climber. Without careful selection, EA acts similar to a hill climber. On the contrary, with 50 and 30 percent selection, EA performs much better than regular EA. In the high selection pressure case (30 %), a high selection pressure method reduces the error by half compared to the regular EA. In the symbolic regression problem, the initial population plays a significant role. Since the crossover and mutation only cause a small change in equations, thus without “good” parents, the EA rarely achieves a desirable fitness. The statement above may be the reason that hill climbers and regular EA perform poorly in this problem.

**Plots**

Chart

Description automatically generated

Figure 3: Overall Results

Chart, scatter chart

Description automatically generated

Figure 3: Complexity vs Mean Error Average for EA with 30% selection pressure

Figure above shows the complexity of the EA with 30% selection pressure. Since the MEA of these points are very close, some points are overlapping with others.

Chart, scatter chart

Description automatically generated

Figure 4: Dot plot for EA with 30% selection pressure

Chart

Description automatically generated

Figure 5: Diversity plot for regular EA

Figure above shows the Regular EA diversity plot with in 4 runs. The plot explains why regular EA does not perform very well. The population is initially to be 300, and the diversity decreases by half and eventually down to be 50; thus, the low diversity does not lead to an optimal result.

Chart

Description automatically generated

Figure 6: Convergence plot for EA with 30% selection pressure

Chart, line chart, box and whisker chart

Description automatically generated

Figure 7: Learning curve for Random Search

Chart, line chart

Description automatically generated

Figure 8: Learning curve for Hill Climber

Chart, line chart

Description automatically generated

Figure 9: Learning curve for Regular EA

Chart, line chart

Description automatically generated

Figure 10: Learning curve for EA 30%

Chart, line chart

Description automatically generated

Figure 11: Learning curve for EA 50%

**Appendix**

Random Search

import random

from collections import deque

from numba import jit

import numpy as np

from copy import deepcopy

iteration = 10000

time = 1

best = float('inf')

MEA\_record = []

f = open("data.txt")

line = f.readline()

x = []

y = []

while line:

  num = list(map(float,line.split(',')))

  x.append(num[0])

  y.append(num[1])

  line = f.readline()

x\_train, y\_train = np.array(x[:600]), np.array(y[:600])

x\_valid, y\_valid = np.array(x[600 :]), np.array(y[ 600 :])

def add(a, b):

  return np.add(a, b)

def minus(a,b):

  return np.subtract(a,b)

def mul(a,b):

  return np.multiply(a,b)

def divd(a ,b):

  return np.divide(a, np.add(b,0.001))

def sin(a):

  return np.sin(a)

def cos(a):

  return np.cos(a)

def constant():

  p = random.random()

  if p > 0.3:

    return float(random.randint(-10, 10))

  else:

    return 'x'

def generate(operator):

  p = random.random()

  if p > 0.35:

    return operator[random.randint(0,5)]

  else:

    return constant()

def createheap(operator):

  depth = 2

  output = []

  cur = 0

  q = deque([operator[random.randint(0,5)]])

  while cur <= depth:

    size = len(q)

    for \_ in range(size):

      node = q.popleft()

      output.append(node)

      if node not in operator:

        q.append(None)

        q.append(None)

      elif cur == depth - 1:

        if node == operator[4] or node == operator[5]:

          q.append('x')

          q.append(None)

        else:

          q.append(constant())

          q.append('x')

      elif node == operator[4] or node == operator[5]:

          left\_node = generate(operator)

          q.append(left\_node)

          q.append(None)

      else:

          left\_node = generate(operator)

          right\_node = generate(operator)

          q.append(left\_node)

          q.append(right\_node)

    cur += 1

    equation = output.copy()

  return output, equation

operator = [add, minus, mul, divd, sin, cos]

population = 300

out,  equ = createheap(operator)

def evaluate(matrix, X, operator):

  for i in range(len(matrix)-1, -1, -1):

    if matrix[i] == 'x':

      matrix[i] = X

    else:

      if matrix[i] == operator[4] or matrix[i] == operator[5]:

        if i == 0:

          matrix[0] = matrix[0](matrix[1])

        else:

          matrix[i] = matrix[i](matrix[2\*i + 1])

      elif matrix[i] == operator[0] or matrix[i] == operator[1] or matrix[i] == operator[2] or matrix[i] == operator[3]:

        if i == 0:

          matrix[0] = matrix[0](matrix[1], matrix[2])

        else:

          matrix[i] = matrix[i](matrix[2\*i + 1], matrix[2\*i + 2])

  return matrix[0]

y\_predict = []

print(equ)

for num in x\_train:

  pred = evaluate(equ.copy(), num, operator)

  y\_predict.append(pred)

Hill Climber

clc

clear

error\_sum = [];

plot\_bar=1:1:10000;

trail=0;

for count1=1:4

trail=trail+1;

[error,x\_initial,y\_initial,y\_actual,F]=main();

error\_sum = [error\_sum;error];

end

range=2500;

out = sum(error\_sum)/4;

CON\_plot=[];

for i=1:4

con\_plot=[];

not=0;

for j = 1:10000

if error\_sum(i,(j))<1.4

con\_plot = [con\_plot (j-not)/10000\*100];

else

not=not+1;

con\_plot = [con\_plot 0];

end

end

CON\_plot=[CON\_plot;con\_plot];

end

CON\_plot=sum(CON\_plot)/10;

figure

plot(plot\_bar,CON\_plot)

title('Convergence value smaller than 1.4')

xlabel('Evals amount')

ylabel('Convergence value (%)')

figure scatter(plot\_bar,error)

title('Hill Climber Dot plot')

xlabel('MAE')

ylabel('Evals amount')

figure scatter(x\_initial,y\_initial)

title('Hill Climber Search Result VS True result')

xlabel('X value')

ylabel('Y value')

hold on

scatter(x\_initial,y\_actual)

legend({'True plot','Hill Climber plot'},'Location','southwest')

figure

plot(out)

%title('Random Search Error VS Evals')

xlabel('Evaluations')

ylabel('MEA')

hold on

for count2 = 1:4

STD = std(error\_sum(:,1+range\*(count2-1)));

error1 = STD/sqrt(4);

x\_bar = 1:range:4\*range;

y\_bar = out(1:range:end);

errorbar(x\_bar(count2),y\_bar(count2),error1);

end

function [initial\_list,Depth] = random(operator)

Depth = 6;

Length = 2^Depth-1;

initial\_list= strings(1,Length);

parent=randperm(8,1);

if operator(parent)=="number"

initial\_list(1)=rand\*20-10;

else

initial\_list(1)=operator(parent);

end

for i = 1:2^(Depth-2)-1

son = randperm(8,1);

if operator(son)=="number"

initial\_list(2\*i)=rand\*20-10;

else

initial\_list(2\*i)=operator(son);

end

son2 = randperm(8,1);

if operator(son2)=="number"

initial\_list(2\*i+1)=rand\*20-10;

else

initial\_list(2\*i+1)=operator(son2);

end

end

for j=2^(Depth-2):2^(Depth-1)-1

son = randperm(2,1)+6;

if operator(son)=="number"

initial\_list(2\*j)=rand\*20-10;

else

initial\_list(2\*j)=operator(son);

end

son2 = randperm(2,1)+6;

if operator(son2)=="number"

initial\_list(2\*j+1)=rand\*20-10;

else

initial\_list(2\*j+1)=operator(son2);

end

end

end

function[new\_list,new\_actualy,new\_result,new\_shorterror]= mutate(initial\_list,Depth,X,Y,short\_error)

check1=initial\_list;

result1 = compute(1,Depth,initial\_list);

Length = 2^(Depth-1)-1;

Change1 = randperm(Length,1);

Change2 = randperm(Length,1);

mid= initial\_list(Change2);

initial\_list(Change2)=initial\_list(Change1);

initial\_list(Change1)=mid;

check2=initial\_list;

c\_initial\_list=correct(initial\_list,Depth);

result2 = compute(1,Depth,c\_initial\_list);

[actual\_y,error]=Finderror(result2,X,Y); %error %short\_error

if error<=short\_error

short\_error=error;

new\_result=result2;

new\_list=c\_initial\_list;

else

mid= initial\_list(Change2);

initial\_list(Change2)=initial\_list(Change1);

initial\_list(Change1)=mid;

[actual\_y,~]=Finderror(result1,X,Y);

new\_result=result1;

new\_list=initial\_list;

end

new\_actualy=actual\_y;

new\_shorterror=short\_error;

end

function [Error,X,Y,Y\_error,F]=main()

data=load('data.txt');

X=data(:,1);

Y=data(:,2);

operator=["+" "-" "\*" "/" "sin" "cos" "x" "number"];

Error=[]; F=[]; counter=0;

[List,depth]=random(operator);

List=correct(List,depth);

result=compute(1,depth,List);

[~,short]=Finderror(result,X,Y);

for i=1:10000

counter=counter+1;

[New\_List,actual\_y,result,error] = mutate(List,depth,X,Y,short);

List=New\_List;

short=error;

Y\_error=actual\_y;

F=[F result];

Error=[Error short];

disp(i)

end

end

function [actual\_y,error]=Finderror(result,X,Y)

sum=0;

actual\_y=[];

for i =1:1000

x=X(i);

Con=eval(result);

actual\_y=[actual\_y;Con];

diff=abs(Con-Y(i));

sum=sum+diff;

end

a=1;

error=sum/1000;

end

function correct\_list=correct(initial\_list,Depth)

for i=1:2^(Depth-1)-1

if initial\_list(i)=="-"&&initial\_list(2\*i)=="x"&&initial\_list(2\*i+1)=="x"

initial\_list(2\*i+1)=rand\*20-10;

end

end

correct\_list=initial\_list;

end

function result = compute(level,Depth,initial\_list)

result=strings(1,1);

if level>=2^(Depth-2)

if (str2double(initial\_list(level))<100)==1 || initial\_list(level)=="x"

result = result + initial\_list(level);

elseif initial\_list(level)=="sin" || initial\_list(level)=="cos"

result = result + initial\_list(level)+"("+initial\_list(2\*level)+")";

else

result = result+"("+initial\_list(2\*level)+initial\_list(level)+initial\_list(2\*level+1)+")";

end

else

if (str2double(initial\_list(level))<20)==1 || initial\_list(level)=="x"

result = result + initial\_list(level);

elseif initial\_list(level)=="sin" || initial\_list(level)=="cos"

result = result + initial\_list(level)+"("+compute(2\*level,Depth,initial\_list)+")";

else

result = result+"("+compute(2\*level,Depth,initial\_list)+initial\_list(level)+compute(2\*level+1,Depth,initial\_list)+")";

end

end

end

EA 30%

import random

from collections import deque

import numpy as np

iteration = 1

time = 1

best = float('inf')

MEA\_record = []

f = open(r"data.txt")

line = f.readline()

x = []

y = []

while line:

    num = list(map(float,line.split(',')))

    x.append(round(num[0],5))

    y.append(round(num[1],5))

    line = f.readline()

f.close()

x\_train, y\_train = np.array(x), np.array(y)

#x\_valid, y\_valid = np.array(x[600 :]), np.array(y[ 600 :])

def constant():

    p = random.random()

    if p > 0.3:

        return float(random.randint(-10, 10))

    else:

        return 'x'

def add(a, b):

    return np.add(a, b)

def minus(a,b):

    return np.subtract(a,b)

def mul(a,b):

    return np.multiply(a,b)

def divd(a ,b):

        return np.divide(a, np.add(b,0.001))

def sin(a):

    return np.sin(a)

def cos(a):

    return np.cos(a)

operator = [add, minus, mul, divd, sin, cos]

population = 300

def fitness(a, b):

    difference = np.subtract(a,b)

    fitness = np.absolute(difference)

    total = np.sum(fitness, axis = 0)

    avg = np.divide(total, 600)

    return round(avg,6)

def generate(operator):

    p = random.random()

    if p > 0.35:

        return operator[random.randint(0,5)]

    else:

        return constant()

def createheap(operator):

    depth = 5

    output = []

    cur = 0

    q = deque([operator[random.randint(0,5)]])

    while cur <= depth:

        size = len(q)

        for \_ in range(size):

            node = q.popleft()

            output.append(node)

            if node not in operator:

                q.append(None)

                q.append(None)

            elif cur == depth - 1:

                if node == operator[4] or node == operator[5]:

                    q.append('x')

                    q.append(None)

                else:

                    q.append(constant())

                    q.append('x')

            elif node == operator[4] or node == operator[5]:

                left\_node = generate(operator)

                q.append(left\_node)

                q.append(None)

            else:

                left\_node = generate(operator)

                right\_node = generate(operator)

                q.append(left\_node)

                q.append(right\_node)

        cur += 1

        equation = output.copy()

    return output,  equation

def findconstant(matrix):

    candidate = []

    for i in range(len(matrix)):

        if type(matrix[i]) == float:

            candidate.append(i)

    return candidate

def mutation(matrix, candidate, operator, t):

    limit = 3

    for \_ in range(limit):

        if len(candidate) != 0:

            for i in range(len(candidate)):

                idx = candidate[i]

                matrix[idx] += random.random()\* random.randint(-10, 10)

        idx = random.randint(0, 40)

        while matrix[idx] not in operator:

            idx = random.randint(0, 40)

        if matrix[idx] in operator[:4]:

            new = random.randint(0, 3)

            matrix[idx] = operator[new]

        elif matrix[idx] in operator[4:]:

            new = random.randint(4, 5)

            matrix[idx] = operator[new]

    return  matrix

def evaluate(matrix, x\_train, operator):

    for i in range(len(matrix)-1, -1, -1):

        if matrix[i] == 'x':

            matrix[i] = x\_train

        else:

            if matrix[i] == operator[4] or matrix[i] == operator[5]:

                if i == 0:

                    matrix[0] = matrix[0](matrix[1])

                else:

                    matrix[i] = matrix[i](matrix[2\*i + 1])

            elif matrix[i] == operator[0] or matrix[i] == operator[1] or matrix[i] == operator[2] or matrix[i] == operator[3]:

                if i == 0:

                    matrix[0] = matrix[0](matrix[1], matrix[2])

                else:

                    matrix[i] = matrix[i](matrix[2\*i + 1], matrix[2\*i + 2])

    return matrix[0]

def find\_crossingidx(matrix1, matrix2, operator):

    for j in range(len(matrix1) - 1, -1, -1):

        if matrix1[j] in operator:

            function1\_idx = j

            break

    for i in range(len(matrix2) - 1, -1, -1):

        if matrix2[i] in operator:

            function2\_idx = i

            break

    return [function1\_idx, function2\_idx]

def crossing(i, j, matrix1, matrix2):

    children = matrix1.copy()

    if children[i] not in operator[4:] and matrix2[j] not in operator[4:]:

        children[i], children[2\*i + 1], children[2\*i + 2] = matrix2[j], matrix2[2\*j + 2], matrix2[2 \*j +1]

    else:

        children[i], children[2\*i + 1], children[2\*i + 2] = matrix2[j], matrix2[2\*j + 1], matrix2[2 \*j + 2]

    return children

def rank(dic, y\_train, x\_train, operator):

    rank\_dic = []

    output\_dic = []

    totalpoint = []

    for i in range(600):

        result = evaluate(dic[i].copy(), x\_train, operator)

        MEA = fitness(y\_train, result)

        rank\_dic.append([MEA, dic[i]])

        totalpoint.append(MEA)

    rank\_dic = sorted(rank\_dic, key = lambda x: x[0])

    count = 0

    for j in range(population):

        output\_dic.append(rank\_dic[j][1])

        if rank\_dic[j][0] <= 0.8:

            count += 1 / 300

    return output\_dic, rank\_dic[0][0], round(count, 3), totalpoint

initial\_dictionary = []

for i in range(population):

    output, equation = createheap(operator)

    initial\_dictionary.append(output)

#print(dictionary[0])

#print('-----------------------------------------------------------')

#print(dictionary)

#print('---------------------------------------------------------------')

for \_ in range(1):

    cur = []

    valid = []

    convergence = []

    dictionary = initial\_dictionary.copy()

    complexityList = []

    dotList = []

    for t in range(10000):

        for \_ in range(population//3):

            idx\_0 = random.randint(0, 199)

            idx\_1 = random.randint(0, 299)

            i , j = find\_crossingidx(dictionary[ idx\_0 ], dictionary[ idx\_1 ], operator)

            first\_children = crossing(i, j, dictionary[idx\_0], dictionary[ idx\_1 ])

            i2 , j2 = find\_crossingidx(dictionary[ idx\_1 ], dictionary[ idx\_0 ], operator)

            second\_children = crossing(i2, j2, dictionary[ idx\_1 ], dictionary[ idx\_0 ])

            candidate0 = findconstant(first\_children)

            candidate1 = findconstant(second\_children)

            mutated0 = mutation(first\_children, candidate0, operator,t)

            mutated1 = mutation(second\_children, candidate1, operator, t)

            dictionary.append(mutated0)

            dictionary.append(mutated1)

        for \_ in range(100):

            fresh, fresh\_equation = createheap(operator)

            dictionary.append(fresh)

        dictionary, Best\_MEA, propotion, dot = rank(dictionary, y\_train, x\_train, operator)

        #if t in [0, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000]:

        #    result = evaluate(dictionary[0].copy(), x\_valid, operator)

        #    MEA\_valid = fitness(y\_valid, result)

        #    valid.append(MEA\_valid)

        dotList.append(dot)

        cur.append((Best\_MEA))

        convergence.append(propotion)

        empty = 0

        for slot in dictionary[0].copy():

            if slot == None:

                empty+=1

        complexity = len(dictionary[0].copy()) - empty

        complexityList.append(complexity)

        print('evol : {},  MEA is {},  convergence : {}'.format(t, Best\_MEA, propotion))

    MEA\_record.append(cur)

        #print(dictionary)

        #print(sorted\_order)

        #print('------------------------------------------------------')

    np.savetxt("MAE.mat", np.array(cur), fmt="%s")

#print(MEA\_record)

#print(dictionary[0])

EA\_RANK = evaluate(dictionary[0].copy(), x\_train, operator)

#print(MEA\_record)

#print(predict\_y[:5])

np.savetxt("EA\_rank\_final.mat", np.array(EA\_RANK), fmt="%s")

np.savetxt("convergence.mat", np.array(convergence), fmt="%s")

# In[42]:

import matplotlib.pyplot as plt

# In[43]:

plt.scatter(x\_train, y\_train)

plt.scatter(x\_train, EA\_RANK)

# In[58]:

x = [i for i in range(10000)]

plt.plot(x, cur)

# In[48]:

x = []

y = []

for i in range(len(dotList)):

    for j in range(len(dotList[i])):

        x.append(i)

        y.append(dotList[i][j])

# In[55]:

plt.ylim([0, 100000])

plt.scatter(x, y, s=3.0)

# In[50]:

plt.scatter(complexityList, cur, s=10)