COLUMBIA UNIVERSITY

MECE 4510 EVOLUTIONARY COMPUTATION AND DESIGN AUTOMATION

Symbolic Regression

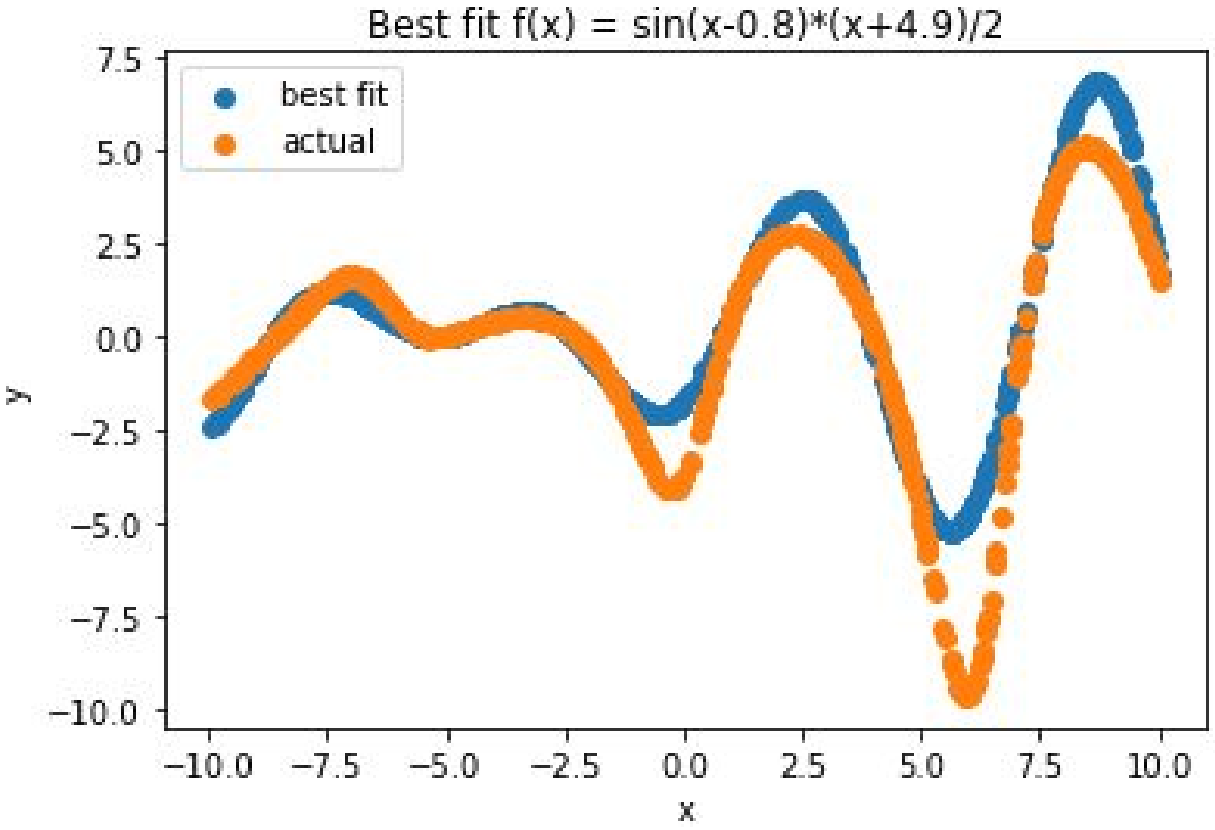
Yifan Gui
UNI: yg2751

Instructor:
Dr. Hod Lipson

Grace Hour Used: 0
Grace Hour Gained: 1
Grace Hour Remaining: 93

Oct 20, 2020

Result Summary



The best fitting Symbolic Regression function is:

$$f(x) = sin(x - 0.8) * (x + 4.9)/2$$

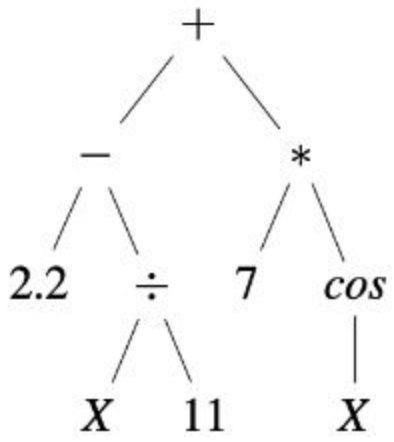
Method	Evaluation Number	MAE
Random Search	300000	1.5511
Hill Climber	300000	0.6988
GP	30000	0.6935
GP(random mutation rate)	30000	0.6742
GP(incremental mutation rate)	15000	0.6919

Method

For this assignment, we are expected to use genetic programming to make a symbolic regression with 1000 data points provided. We need to express the symbolic algebraic expression in the form of $y = f(x)$ which is able to handle mathematical operations likes $+$, $-$ $*$, $/$, *sine*, *cosine* and real constant in range of ± 10 with the variable x.

Representation

Tree structure is used in the assignment to store data points and mathematical operations for computing. The benefit of using tree structure is that it is easier to do a recursive evaluation. The tree structure can be represented as following:



It represents:

$$2.2 - (X/11) + 7 * cosX$$

In this assignment, the maximum depth of tree structure is assumed to be less or equal than 6.

Random Search

This method is straight forward. For each iteration, it generates a function tree with depth varies from 3 to 6. Since there is no selection involved in the method, results generated randomly and mean absolute error doesn't change a lot for a limited amount of iterations. Random search is also the base logic for hill climber and genetic programming.

Hill Climber

The Hill Climber involves some decision making. It compares the MAEs (mean absolute error) from the new tree and the old tree, if the new one is smaller, it updates to the new one. However, since the starting point of this method is randomly selected, it can just find a local optimal solution instead of the one for overall.

Genetic Programming with Variations

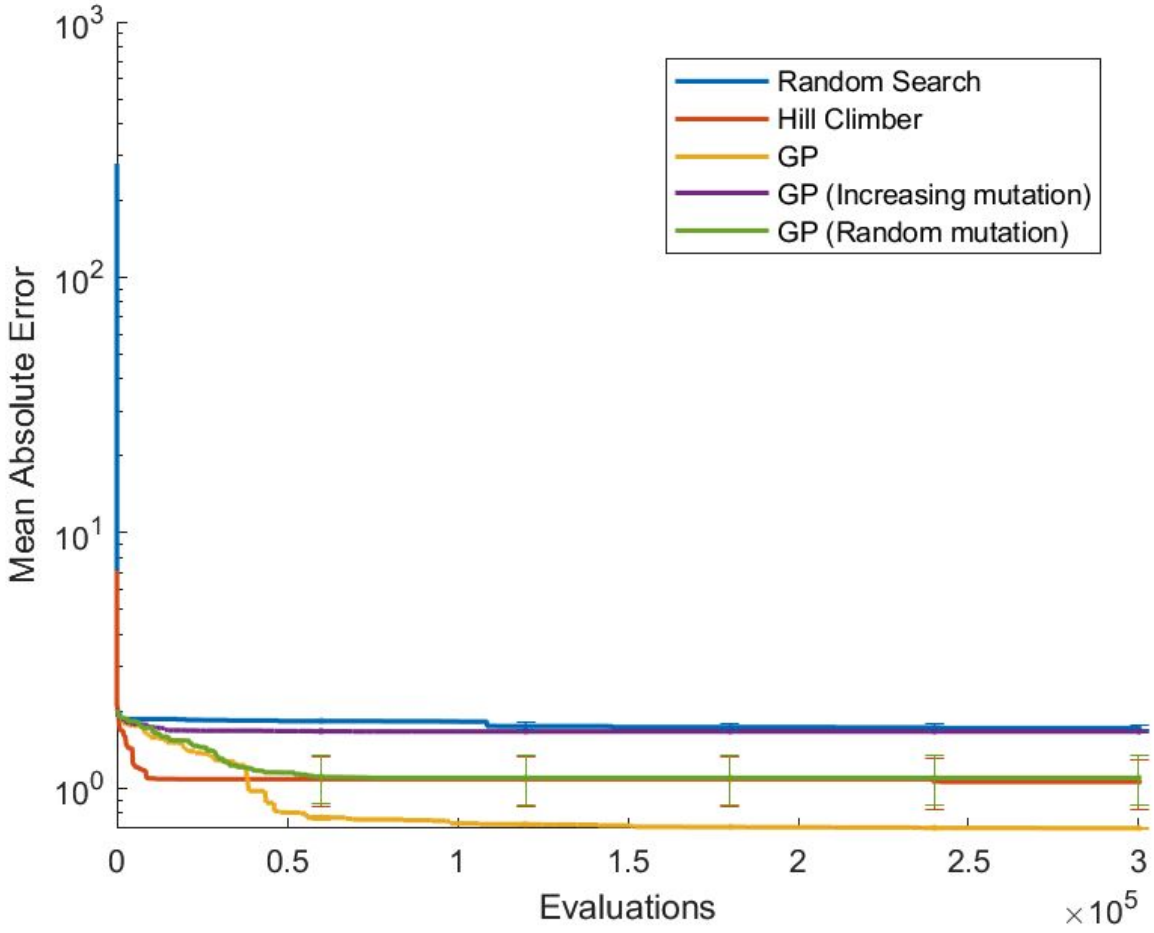
Genetic Programming consists of selection, crossover and mutation. The method selects optimal solutions from initial generation and uses them to breed offspring, which contains a higher percentage of optimal solution path. Variations of GPs are in mutation. Detailed implementation of the method are as following:

- Selection: In the current iteration, select the top 15% best solutions as parents to breed offsprings.
- Crossover: Randomly select a portion of parents' solution to generate offspring, then append solution form one child to another.
- Mutation: there are 3 different mutation type:
    a. Standard mutation: defined mutation rate to be 80%, use this rate to mutate and generate offspring.
    b. Random mutation rate:mutation rate is randomly generated between 0 and 1.
    c. Incremental mutation rate: mutation rate is being incremented by 0.00001.

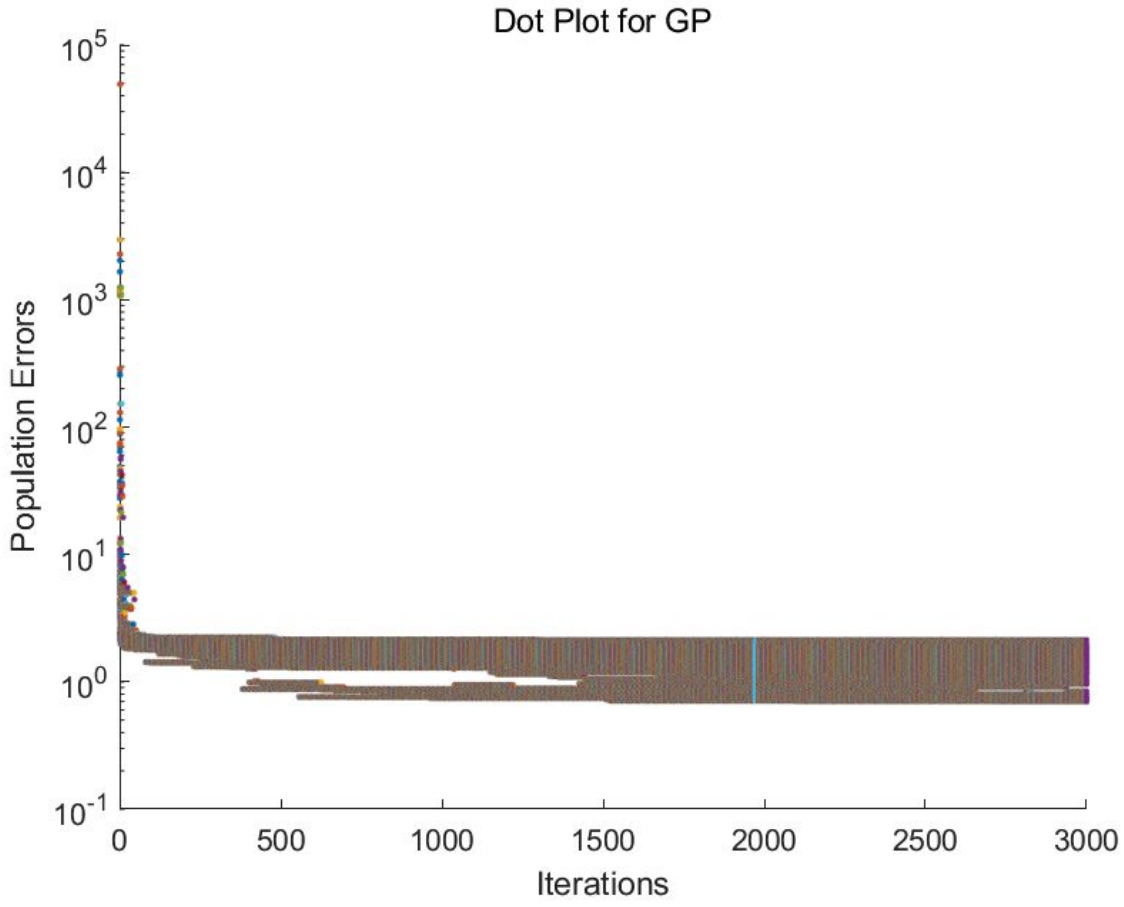
Analysis of Performance

In general, the performance of Genetic Programming methods are much better than other methods. Random search doesn't work well because there is no decision making process in the method, every solution is randomly generated. The Hill Climber method does involve decision making, it compares the new MAE to the old one, and keeps the better solution, however, the drawback of this method is that it can be trapped into a local optimal solution. The GPs are very effective in finding optimal solutions by operating selection, crossover and mutation, which raises the probability of finding optimal solutions.

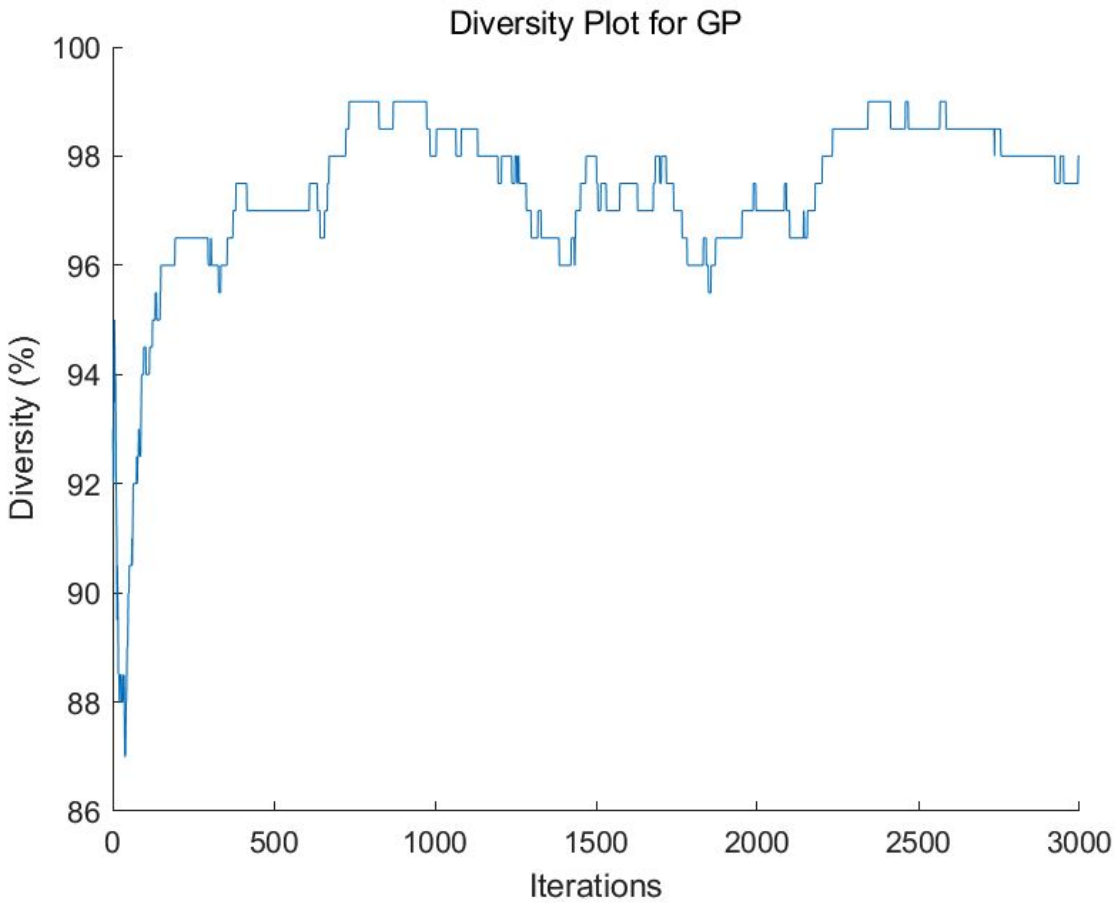
Performance Plots



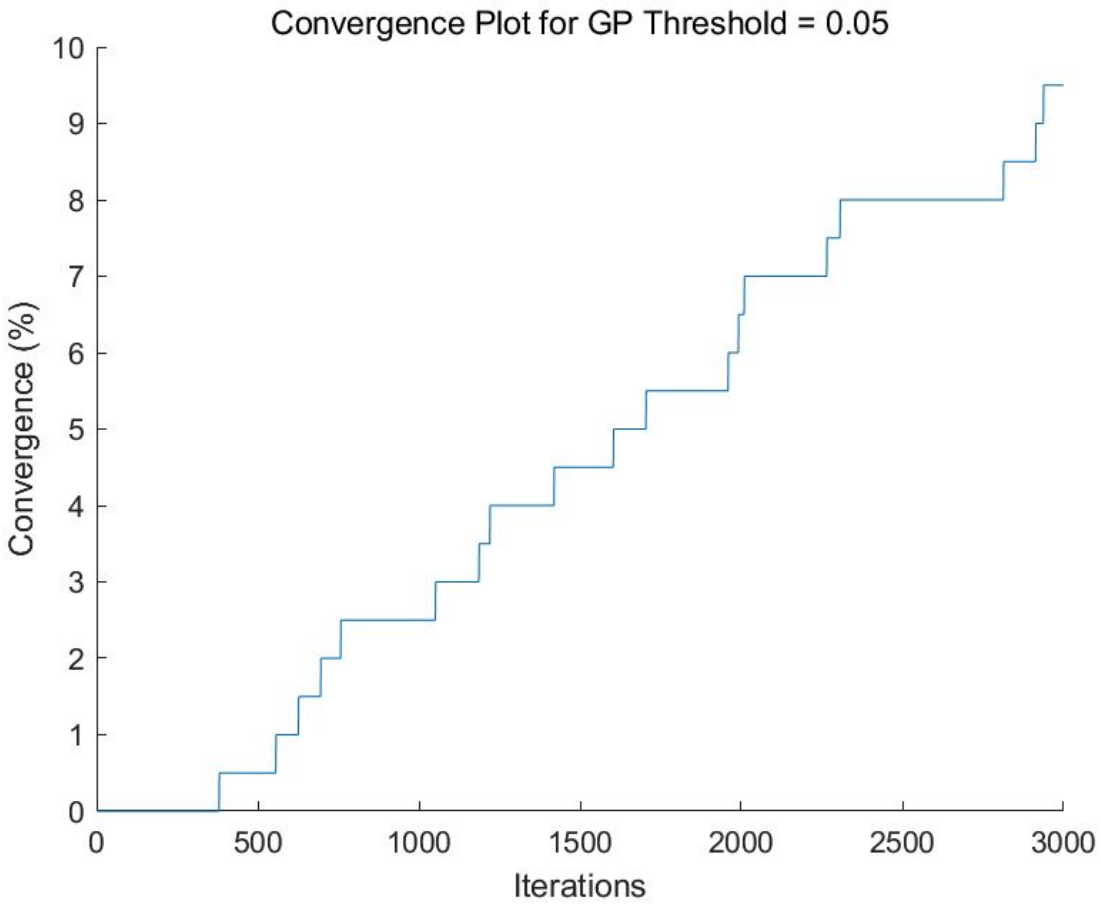
Dot Plot


Dot Plot for GP
Population Errors
Iterations
10^5
10^4
10^3
10^2
10^1
10^0
10^{-1}
0
500
1000
1500
2000
2500
3000

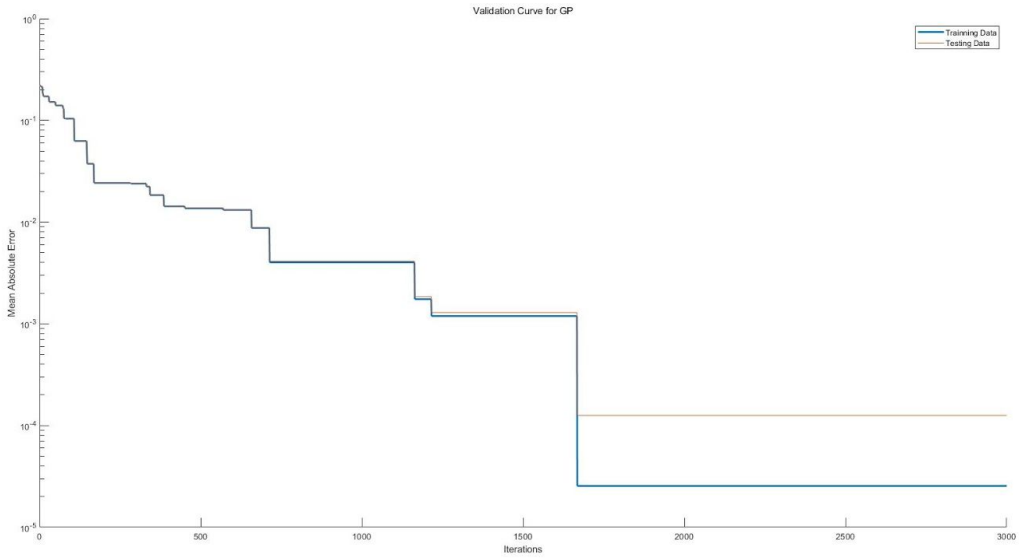
Diversity Plot



Convergence Plot



Validation


Validation Curve for GP
Trainning Data
Testing Data
Mean Absolute Error
Iterations
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
0
500
1000
1500
2000
2500
3000

Appendix

```
import random
import math
import sys
from copy import deepcopy
import matplotlib.pyplot as plt
from math import *
import numpy as np
import seaborn as sns

unary = ["sin", "cos"]
binary = ['-', '+', '*','/']

#   Tree parameters
INITIALDEPTH = 4
MAXDEPTH = 4

#   GA global parameters
NBGENERATIONS = 300000
NBTREES = 200
MUTERATE = 0.8
ELITERATE = 0.15
RANKSELECTION = True
ELITISM = True


#   GA transformations parameters
Brate = 0.70 # probability to generate binary operator over unary one.
CSTrate = 0.20 # probability to generate a constant value instead of variable.
VARrate = 0.05
CSTboundaries = 10 # Maximum absolute value of generable constant value.
THRESHOLD = 2
# ------------------------------------------------------------------------

# Nodes and Trees containers
Nodes = {}
Trees = {}
# Nodes and Trees indexes
IdNode = 0
IdTree = 0


# GACLASS ------------------------------------------------------------------- #
class GA:
# Constructor
    def __init__(self, dataset):

        self.nbtrees = NBTREES
        self.t_ids = [IdTree + i for i in range(self.nbtrees)]
        self.t_offsprings = []
        self.t_parents = []
        self.ini_depth = INITIALDEPTH
        for i in range(self.nbtrees):
            generate_tree(self.ini_depth)


        self.dataset = dataset

        self.mute_rate = MUTERATE
        self.elite_rate = ELITERATE
        self.nb_generations = NBGENERATIONS
        self.rankselection = RANKSELECTION
        self.elitism = ELITISM
        self.threshold = THRESHOLD

# Methods
```

```
    def run(self, mode):


        result_train = []
        result_val = []
        iteration = [i for i in range(self.nb_generations)]
        dot_mae = []
        dot_iter = []
        diversity = []
        con_pct = []
        if mode == "ga":

            result_train = []
            result_val = []
            iteration = [i for i in range(self.nb_generations)]
            dot_mae = []
            dot_iter = []
            diversity = []
            con_pct = []
            for i in range(self.nb_generations):
                num = 0
                self.selection()
                self.crossover()
                self.mutation()
                self.output(i)
                result_train.append(Trees[self.t_ids[0]].compute_train_MAE(self.dataset))


                result_val.append(Trees[self.t_ids[0]].compute_val_MAE(self.dataset))
                iter_tree_mae= []
                for j in self.t_ids:
                    if Trees[j].compute_train_MAE(self.dataset) != float("inf"):
                        iter_tree_mae.append(Trees[j].compute_train_MAE(self.dataset))
                    else:
                        iter_tree_mae.append(10)
                    if Trees[j].compute_train_MAE(self.dataset) <= self.threshold:
                        num += 1
                    dot_iter.append(i)
                    dot_mae.append(Trees[j].compute_train_MAE(self.dataset))
                diversity.append(min(iter_tree_mae)/max(iter_tree_mae))
                con_pct.append(num / self.nbtrees)
            """
            # convergence plot
            plt.plot(iteration,con_pct, label = "GA")
            plt.title("convergence plot")
            plt.xlabel("iterations")
            plt.ylabel("convergence(%)")
            plt.legend()
            plt.show()

            # diversity plot
            plt.plot(iteration,diversity, label = "GA")
            plt.xlabel("iterations")
            plt.ylabel("diversity")
            plt.legend()
            plt.show()

            #dot plot
            plt.scatter(dot_iter,dot_mae)
            plt.xlabel("iterations")
            plt.ylabel("population mae")
            plt.show()

            # traning & validation
            plt.plot(iteration,result_train,color = "green", label = "training")
            plt.plot(iteration,result_val,color = "blue", label = "val")
            plt.legend()
            plt.xlabel("iterations")
```

```
            plt.ylabel("mae")
            plt.show()
            """
            return result_train

        elif mode == "hp":

            for i in range(self.nb_generations):
                self.mute_rate = random.random()
                num = 0
                self.selection()
                self.crossover()
                self.mutation()
                self.output(i)
                result_train.append(Trees[self.t_ids[0]].compute_train_MAE(self.dataset))
            return result_train
        elif mode == "im":
            for i in range(self.nb_generations):
                num = 0
                self.mute_rate += 0.00001
                self.selection()
                self.crossover()
                self.mutation()
                self.output(i)
                result_train.append(Trees[self.t_ids[0]].compute_train_MAE(self.dataset))
            return result_train
        else:
            for i in range(1):
                result_train.append(Trees[self.t_ids[0]].compute_train_MAE(self.dataset))
            return Trees[self.t_ids[0]].compute_train_MAE(self.dataset)

        """
        plt.plot(iteration,con_pct)
        plt.plot(iteration,diversity)
        plt.plot(iteration,result_train)
        plt.scatter(dot_iter,dot_mae)


        plt.plot(iteration,result_train,color = "green", label = "training")
        plt.plot(iteration,result_val,color = "blue", label = "val")
        plt.legend()
        plt.xlabel("iterations")
        plt.ylabel("mae")
        plt.show()
        """


        #return Trees[self.t_ids[0]].toRPN(),Trees[self.t_ids[0]].compute_train_MAE(self.dataset)


    def selection(self):
        self.update_fitness()
        sumfit = 0


        if self.rankselection == True:
            for i in range(self.nbtrees):
                Trees[self.t_ids[i]].fitness = self.nbtrees - i


        self.t_offsprings = []
        if self.elitism == True:
            nb_elites = math.ceil(self.nbtrees*self.elite_rate)
            if nb_elites % 2 != 0: nb_elites += 1
            for i in range(nb_elites):
                self.t_offsprings.append(Trees[self.t_ids[i]].copy())
        else:
            nb_elites = 0
```

```
        self.t_parents = []
        sumfit = sum([Trees[i].fitness for i in self.t_ids])
        for i in range(self.nbtrees - nb_elites):
            G = random.uniform(0, sumfit)
            res = 0
            k = 0
            while (res < G):
                res += Trees[self.t_ids[k]].fitness
                k += 1
            self.t_parents.append(Trees[self.t_ids[k-1]].copy())


        for tree_id in self.t_ids:
            Trees[tree_id].del_()
        self.t_ids = []


    def crossover(self):
        newparents = []
        for i in range(len(self.t_parents)//2):
            a = random.choice(self.t_parents)
            b = random.choice(self.t_parents)
            offa = Trees[a].copy()
            offb = Trees[b].copy()
            Trees[offa].crossover(Trees[offb])
            newparents += [offa, offb]


        for tree_id in self.t_parents:
            Trees[tree_id].del_()

        self.t_parents = deepcopy(newparents)


    def mutation(self):
        for i in self.t_parents:
            if random.randint(0,100)/100 < self.mute_rate:
                Trees[i].subtree_mutation()


        self.t_offsprings += self.t_parents
        self.t_ids = deepcopy(self.t_offsprings)


    def update_fitness(self):
        for tid in self.t_ids:
            Trees[tid].update_fitness(self.dataset)
        self.t_ids.sort(key=lambda x: Trees[x].fitness, reverse = True)


    def output(self, i):
        print(i,"MAE : ",Trees[self.t_ids[0]].compute_train_MAE(self.dataset),
"RPN:",Trees[self.t_ids[0]].toRPN(), file=sys.stderr)
        return Trees[self.t_ids[0]].compute_train_MAE(self.dataset)

    def toString(self):
        print("t_ids : ")
        for i in self.t_ids:
            print(Trees[i].MAE, Trees[i].toString())
        print("t_parents : ")
        for i in self.t_parents:
            print(Trees[i].MAE, Trees[i].toString())
        print("t_offsprings : ")
        for i in self.t_offsprings:
            print(Trees[i].MAE, Trees[i].toString())
        print()
```

```
   # def hc(self):
    #    old_tree = Tree[self.t_ids[0]].toRPN()
     #   old_cost


    def hc(self):
        hc_result = []
        old_tree = deepcopy(Trees[self.t_ids[0]].toRPN())
        old_error = deepcopy(Trees[self.t_ids[0]].compute_train_MAE(self.dataset))
        best_error = 10
        for i in range(self.nb_generations):
            Trees[self.t_ids[0]].subtree_mutation()
            new_tree = deepcopy(Trees[self.t_ids[0]].toRPN())
            new_error = deepcopy(Trees[self.t_ids[0]].compute_train_MAE(self.dataset))
            if new_error < old_error:
                old_tree = new_tree
                old_error = new_error
                best_error = new_error
            hc_result.append(best_error)
        return hc_result

# ------------------------------------------------------------------------ #



# TREECLASS ---------------------------------------------------------------- #
class Tree:
    def __init__(self):
        global IdTree
        self.id = IdTree
        Trees[IdTree] = self
        IdTree += 1
        self.MAE = -1

    def del_(self):
        Nodes[self.root].del_()
        del Trees[self.id]

    def copy(self):
        T = Tree()
        n = Nodes[self.root].copy()
        T.set_root(n)
        T.MAE = self.MAE
        return T.id

    def set_root(self, nid):
        self.root = nid

    def regulate(self):
        Nodes[self.root].regulate(MAXDEPTH)

    def toRPN(self):
        res = RPN("",self.root)
        return res

    def to_list(self):
        return to_list(self.root, [])

    def to_list_parents(self):
        return to_list_parents(self.root, [])

    def update_fitness(self, dataset):
        self.MAE = self.compute_train_MAE(dataset)
        self.fitness = 1/self.MAE

    def evaluate(self, Xi):
        return evaluate_RPN(self.toRPN(), Xi)
```

```
    def compute_train_MAE(self, dataset):
        N = dataset[0]
        M = 600
        train_X = dataset[2]
        train_Y = dataset[3]
        MAE = 0
        try:
            for i in range(M):
                MAE += abs(self.evaluate(train_X[i]) - train_Y[i])
        except:
            return float(0.7)
        return MAE/M

    def compute_val_MAE(self, dataset):
        N = dataset[0]
        M = 200
        val_X = dataset[4]
        val_Y = dataset[5]
        MAE = 0
        try:
            for i in range(M):
                MAE += abs(self.evaluate(val_X[i]) - val_Y[i])
        except:
            return float('inf')
        return MAE/M

    def compute_test_MAE(self, dataset):
        N = dataset[0]
        M = 200
        test_X = dataset[6]
        test_Y = dataset[7]
        MAE = 0
        try:
            for i in range(M):
                MAE += abs(self.evaluate(test_X[i]) - test_Y[i])
        except:
            return float('inf')
        return MAE/M



    def crossover(self, tree2):
        id1 = random.choice(self.to_list_parents())
        id2 = random.choice(tree2.to_list_parents())
        child1 = random.choice(Nodes[id1].children)
        child2 = random.choice(Nodes[id2].children)
        Nodes[id1].children.remove(child1)
        Nodes[id2].children.remove(child2)
        Nodes[id1].add_children(child2)
        Nodes[id2].add_children(child1)

    def subtree_mutation(self):
        l = self.to_list()
        nid = random.choice(l)
        d = Nodes[nid].depth()
        for child in Nodes[nid].children:
            Nodes[child].del_()
        Nodes[nid].generate_children(d - 1)
        self.regulate()

    def toString(self):
        return "root = " + str(self.root) + " : " + self.toRPN()
# ------------------------------------------------------------------------ #



# NODECLASS ------------------------------------------------------------- #
```

```
class Node:
# Constructor
    def __init__(self, depth):
        global IdNode
        self.nid = IdNode
        self.var_rate = VARrate
        self.cst_bound = CSTboundaries
        self.cst_rate = CSTrate
        self.bin_rate = Brate

        if (depth == 1) | (random.randint(0,100)/100 < self.var_rate):
            self.type = "unary"
            if random.randint(0,100)/100 < self.cst_rate:
                self.data = round(random.uniform(-self.cst_bound, self.cst_bound),3) # remain 3 digits
after decimal point
            else:
                self.data = "x" + str(random.randint(1, dataset[0]-1))
        else:
            if random.randint(0,100)/100 < self.bin_rate:
                self.type = "binary"
                self.data = random.choice(binary)
            else:
                self.type = "unary"
                self.data = random.choice(unary)

        self.children = []
        Nodes[IdNode] = self
        IdNode += 1

    def del_(self):
        for child in self.children:
            Nodes[child].del_()
        del Nodes[self.nid]

    def copy(self):
        global IdNode
        n = deepcopy(self)
        n.nid = IdNode
        IdNode += 1
        ch = []
        for child in n.children:
            ch.append(Nodes[child].copy())
        Nodes[n.nid] = n
        n.children = ch
        return n.nid

    def generate_children(self, depth):
        self.children = []
        if depth > 0:
            self.leaf = False
            child1 = Node(depth)
            self.add_children(child1.nid)

            if self.type == "binary":
                child2 = Node(depth)
                self.add_children(child2.nid)

            for child in self.children:
                Nodes[child].generate_children(depth-1)
        else:
            self.leaf = True

    def add_children(self, nid):
        self.children.append(nid)

    def regulate(self, depth):
        if depth == 0:
            self.del_()
```

```
        elif depth == 1:
            for child in self.children:
                Nodes[child].regulate(0)
            self.children = []
        else:
            for child in self.children:
                Nodes[child].regulate(depth-1)

    def depth(self):
        if self.isleaf() == True:
            return 1
        else:
            return 1 + max([Nodes[child].depth() for child in self.children])

    def isleaf(self):
        if len(self.children) == 0:
            return True
        return False

    def toString(self):
        res = " "
        res += str(self.data)
        res += " "
        if self.type == "unary":
            res += "U "
        else:
            res += "B "
        res += "childr : " + str(self.children) + ""
        return res
# ------------------------------------------------------------------------ #



# FONCTIONS ------------------------------------------------------------- #
def input(filename):
    f = open(filename)
    n = 2
    m = 0
    DataX = []
    DataY = []
    line = f.readline().strip('\n').split(" ")
    while line != ['']:
        DataX.append(list(map(float,line[:n-1])))
        DataY.append(float(line[-1]))
        line = f.readline().strip('\n').split(" ")
        m += 1
    train_X = DataX[0:600]
    train_Y = DataY[0:600]
    val_X = DataX[600:800]
    val_Y = DataX[600:800]
    test_X = DataX[800:1000]
    test_Y = DataY[800:1000]
    return n, m, train_X, train_Y, val_X, val_Y, test_X, test_Y

def evaluate_RPN(expression, datasetX):
    try:
        stack = []
        for val in expression.split(' '):
            if val in binary:
                op1 = stack.pop()
                op2 = stack.pop()
                if val=='-': r = op2 - op1
                elif val=='+': r = op2 + op1
                elif val=='*': r = op2 * op1
                elif val=='/': r = op2 / op1
                else: r = op2**op1
                stack.append(r)
            elif val in unary:
```

```
                op = stack.pop()
                if val=="~": r = - op
                elif val=="abs": r = abs(op)
                else: r = eval("math."+val+"("+ str(op) +")")
                stack.append(r)
            elif val[0] == "x":
                stack.append(datasetX[int(val[1:])-1])
            else:
                stack.append(float(val))
        return stack.pop()
    except:
        return float('inf')

def RPN(r, nid):
    if Nodes[nid].isleaf() == True:
        return str(Nodes[nid].data)
    if Nodes[nid].type == "binary":
        res = RPN(r, Nodes[nid].children[0]) + " "
        res += RPN(r, Nodes[nid].children[1]) + " "
        res += str(Nodes[nid].data)
        return res
    res = RPN(r, Nodes[nid].children[0]) + " "
    res += str(Nodes[nid].data)
    return res

def to_list(nid, list_=[]):
    list_ += [nid]
    for child in Nodes[nid].children:
        to_list(child, list_)
    return list_

def to_list_parents(nid, list_=[]):
    if Nodes[nid].isleaf() == False:
        list_ += [nid]
        for child in Nodes[nid].children:
            to_list_parents(child, list_)
    return list_

def generate_tree(depth, nid = None):
    global IdTree
    T = Tree()
    if nid == None:
        n = Node(depth)
        nid = n.nid
    T.set_root(n.nid)
    Nodes[n.nid].generate_children(depth-1)
    return nid

def toString_N(Nodes):
    for k, v in Nodes.items():
        print(str(k) + v.toString())

def toString_T(Trees):
    for k, v in Trees.items():
        print(str(k) + " root = " + str(v.root))


# ------------------------------------------------------------------------- #



csvfilename = "SR_div_1000.txt"

dataset = input(csvfilename)


g = GA(dataset)
#GA_output = g.run("ga")
```

```
"""
GA_output = []
for i in range(3):
    g = GA(dataset)
    GA_output.append(g.run("ga"))
"""


"""
g = GA(dataset)
HP_output = g.run("hp")

g = GA(dataset)
IM_output = g.run("im")

g = GA(dataset)
RM_output = []
for i in range(50):
    random.seed(i)
    RM_output.append(g.run("rd"))

hc_output = []
g = GA(dataset)
hc_output = g.hc()

plt.plot(range(1000), hc_output,label = "hc")
plt.plot(range(1000), GA_output,label = "GA")
plt.plot(range(1000), HP_output,label = "GA_HP")
plt.plot(range(1000), IM_output,label = "GA_IM")
plt.plot(range(1000), RM_output,label = "Random search")
plt.legend()
plt.xlabel("iterations")
plt.ylabel("mae")
plt.show()
"""

fit = []
x = []

for i in dataset[2]:
    i = float(i[0])
    fit.append(math.sin(-0.8+i)*(i+4.9)/2)
    x.append(i)


plt.scatter(x,fit,label ="best fit")
plt.scatter(x,dataset[3],label ="actual")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.title("Best fit f(x) = sin(x-0.8)*(x+4.9)/2")
```