**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

**import** **matplotlib.pyplot** **as** **plt**

**from** **tqdm.notebook** **import** tqdm

**from** **sklearn.metrics** **import** mean\_squared\_error

**import** **operator**

**import** **random**

**from** **random** **import** randint,seed

seed(**0**)

**import** **matplotlib.pyplot** **as** **plt**

data = np.genfromtxt("data.txt",delimiter=",")

df = pd.DataFrame(data,columns=["x","y"])

feature\_names = ['x']

target\_name = 'y'

X = df[feature\_names]

y = df[target\_name]

**def** **div**(a, b):

**return** a / b **if** b **else** a

**def** **cos**(a):

**return** np.cos(a)

**def** **sin**(a):

**return** np.sin(a)

**def** **exp2**(a):

**return** a\*\***2**

**def** **exp3**(a):

**return** a\*\***3**

**def** **generate\_function**(depth):

**if** randint(**0**, **10**) >= depth\***2**:

oper = operations[randint(**0**, len(operations) - **1**)]

**return** {

"func": oper["func"],

"children": [generate\_function(depth + **1**) **for** \_ **in** range(oper["arg\_count"])],

"format\_str": oper["format\_str"],

}

**else**:

**return** {"feature\_name": features[randint(**0**, len(features) - **1**)]}

**def** **string\_of\_function**(node):

**if** "children" **not** **in** node:

**return** node["feature\_name"]

**return** node["format\_str"].format(\*[string\_of\_function(c) **for** c **in** node["children"]])

operations = (

{"func": operator.add, "arg\_count": **2**, "format\_str": "({} + {})"},

{"func": operator.sub, "arg\_count": **2**, "format\_str": "({} - {})"},

{"func": operator.mul, "arg\_count": **2**, "format\_str": "({} \* {})"},

{"func": div, "arg\_count": **2**, "format\_str": "({} / {})"},

{"func": cos, "arg\_count": **1**, "format\_str": "np.cos({})"},

{"func": sin, "arg\_count": **1**, "format\_str": "np.sin({})"},

{"func": exp2, "arg\_count": **1**, "format\_str": "({} \*\* 2)"},

{"func": exp3, "arg\_count": **1**, "format\_str": "({} \*\* 3)"}

)

features = ['x',**1**,**2**,**3**,**4**,**5**,**6**,**7**,**8**,**9**,**10**,-**1**,-**2**,-**3**,-**4**,-**5**,-**6**,-**7**,-**8**,-**9**,-**10**]

#Random Search

#First we generate a random function using the given set of operators and variables. Then we fit the x values in that function to obtain a list of y values. We find the Mean Squared Error between calculated y and given y values.We obtain the fitness using 1/MSE. Check if fitness has improved and if so, append.

best\_fitness = **0**

fitness\_evolution\_list = []

**for** epoch **in** tqdm(range(**1000**)):

eq = str(string\_of\_function(generate\_function(**0**))).replace(" ","")

y\_pred = []

X.reset\_index(drop=True)

**for** i **in** range(len(X)):

x = X.iloc[i]

**try**:

pred = float(eval(eq))

**except** (**SyntaxError**, **NameError**, **TypeError**, **ZeroDivisionError**):

pred = **0.0**

**if** type(pred)!=float:

pred = pred['x']

**if** pred == np.inf **or** pred == -np.inf **or** np.isnan(pred):

pred = **0**

y\_pred.append(pred)

fitness = **1**/mean\_squared\_error(y, y\_pred)

**if** fitness > best\_fitness:

best\_fitness = fitness

best\_fit\_function = eq

fitness\_evolution\_list.append((epoch,eq,fitness,y\_pred))

**print**("Epoch "+str(epoch)+": "+str(best\_fit\_function)+"**\n**Fitness :"+str(fitness),end= "**\r**")

#For Hill Climber, we will use Mutation where we replace one node in the current function with a randomly generated function. We check if the fitness is better. If it is, we use the new function as the starting point for next mutation. If not, we continue with the old function. If the fitness doesn't improve after n such iterations, we discard the function and generate a new one for further iterations since continuing with the same function will increase the function complexity and computing time unnecessarily.

**def** **node\_to\_mutate**(function,parent,depth):

**if** "children" **not** **in** function:

to\_mutate = parent

**elif** randint(**0**,**10**)< depth\***2**:

to\_mutate = function

**else**:

count\_of\_subnodes = len(function['children'])

to\_mutate = node\_to\_mutate(function['children'][randint(**0**, count\_of\_subnodes - **1**)],function,depth)

**return** to\_mutate

**def** **mutate**(function):

mutated\_specimen = function

mutation\_node = node\_to\_mutate(function,None,**0**)

total\_subnodes = len(mutation\_node['children'])

mutation\_node["children"][randint(**0**,total\_subnodes-**1**)] = generate\_function(**2.5**)

**return** mutated\_specimen

#Hill Climber Run

best\_fitness = **0**

fitness\_evolution\_list = []

init\_func = generate\_function(**0**)

reset\_count = **0**

**for** epoch **in** tqdm(range(**10000**)):

new\_func = mutate(init\_func)

eq = str(string\_of\_function(new\_func)).replace(" ","")

y\_pred = []

X.reset\_index(drop=True)

**for** i **in** range(len(X)):

x = X.iloc[i]

**try**:

pred = float(eval(eq))

**except** (**SyntaxError**, **NameError**, **TypeError**, **ZeroDivisionError**,**OverflowError**):

pred = **0.0**

**if** type(pred)!=float:

pred = pred['x']

**if** pred == np.inf **or** pred == -np.inf **or** np.isnan(pred):

pred = **0**

y\_pred.append(pred)

reset\_count +=**1**

fitness = **1**/mean\_squared\_error(y, y\_pred)

**if** fitness > best\_fitness:

best\_fitness = fitness

best\_fit\_function = new\_func

init\_func = new\_func

reset\_count = **0**

fitness\_evolution\_list.append((epoch,eq,fitness,y\_pred))

**print**("Best Fitness : " + str(best\_fitness),end = "**\r**")

**if** reset\_count>**10**:

init\_func = generate\_function(**0**)

reset\_count = **0**

#Genetic Programming

#Here, we are using crossover mechanism as a variation method. We start with an initial pool of equations that are randomly generated.

#We find the fitness values of all these functions using Mean Square Error inverse. We random select a small subset of this entire pool to create the mating pool. There we choose the function with the max fitness to choose the first parent. Same is repeated to choose the second parent. Then we randomly choose one node in parent 1 and replace it with another random node from parent 2. We do this repeatedly till we create a newer pool for testing the fitness again. This entire cycle represents one generation. We chose to run it for 10-100 generations based on the initial pool sizes we used.

**def** **select\_parent**(pop,fitness):

random\_members = [randint(**0**,pop\_size-**1**) **for** member **in** range(pool\_size)]

**return** min([(fitness[member], pop[member]) **for** member **in** random\_members],key = **lambda** member: member[**0**])[**1**]

**def** **crossover**(pop,fitness):

parent\_1 = select\_parent(pop,fitness)

parent\_2 = select\_parent(pop,fitness)

offspring = parent\_1

node\_1 = node\_to\_mutate(offspring,None,**0**)

node\_2 = node\_to\_mutate(parent\_2,None,**0**)

total\_subnodes = len(mutation\_node['children'])

node\_1['children'][randint(**0**,total\_subnodes-**2**)] = node\_2

**return** offspring

pop\_size = **300** #Found 300 to be optimum

pool\_size = **50** #For parent selection

population = [generate\_function(**1**) **for** \_ **in** range(pop\_size)]

generations = **10**

output = []

best\_fitness = **0**

**for** gen **in** tqdm(range(generations)):

fitness\_list = []

**for** specimen **in** tqdm(population):

y\_pred = []

X.reset\_index(drop=True)

eq = str(string\_of\_function(specimen)).replace(" ","")

**for** i **in** range(len(X)):

x = X.iloc[i]

**try**:

pred = float(eval(eq))

**except** (**SyntaxError**, **NameError**, **TypeError**, **ZeroDivisionError**,**OverflowError**,**RuntimeError**,**RuntimeWarning**,):

pred = **0.0**

**if** type(pred)!=float:

pred = pred['x']

**if** pred == np.inf **or** pred == -np.inf **or** np.isnan(pred):

pred = **0**

y\_pred.append(pred)

fitness = **1**/(mean\_squared\_error(y, y\_pred)

fitness\_list.append(fitness)

**if** fitness > best\_fitness:

best\_fitness = fitness

best\_prog = specimen

output.append((gen, specimen, fitness, y\_pred))

population = [crossover(population, fitness\_list) **for** \_ **in** range(pop\_size)]