EVOLUTIONARY COMPUTATION & DESIGN AUTOMATION
(MECS – 4510)

Assignment 2: Symbolic Regression

Assignment Report Submitted by

PAVAN KASHINATH KAMATH
UNI: pkk2123

SACHIN FRANCIS DSOUZA
UNI: sfd2121

Instructor Name
PROF. HOD LIPSON

Date Submitted
October-27-2021

Grace hours used: 10

Grace hours remaining: 86

Department of Mechanical Engineering

COLUMBIA | ENGINEERING
The Fu Foundation School of Engineering and Applied Science

Table of Contents

List of Abbreviations

Sl.No.	Abbreviation	Full Form
1.	RS	Random Search
2.	RMHC	Randomized Hill Climber
3.	GP	Genetic Programming

Result Summary Random Search:

```
Best function for Random Search : (np.cos(np.cos(-6))*x)
Best fitness for Random Search : 0.16820722346745326
```

MSE: 5.945055

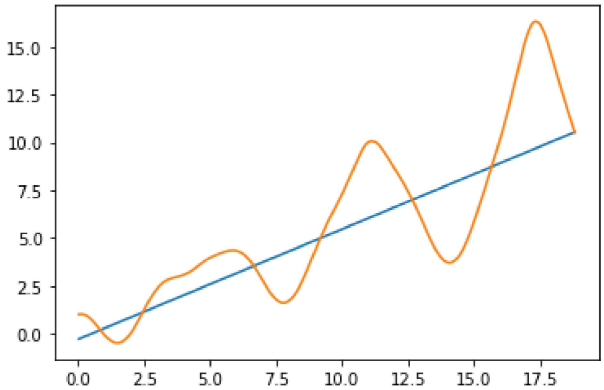

Fig 1: Plotting the Best Fit Curve Random Search.

Results Summary Hill Climber:

```
Best Function is : (((x-np.sin(x))**3)/((x*(9/(3/8)))-6))
```

MSE = 3.64
FITNESS = 0.2747

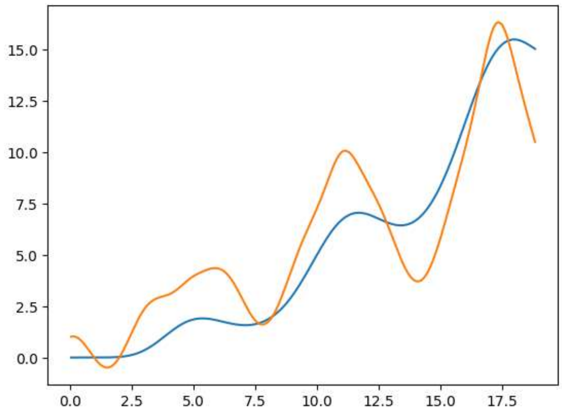

Fig 2: Plotting the Best Fit Curve Hill Climber

Results Summary GP:

```
Best function for GP : (((np.sin((x / x)) ** 3) + (((x - np.sin(((x / x) * x))) - x) * ((x / (x + x)) ** 2))) * (x - np.sin
(x)))
Best fitness for GP : 3.1514596756868816
```

MSE: 0.3173132

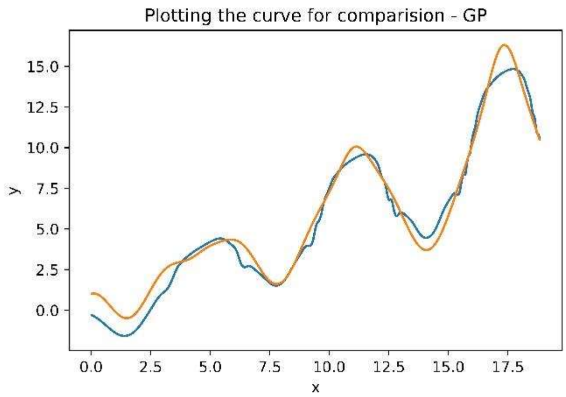

Fig 3: GP Best Fit Curve

Results Summary GP Variation:

```
Best function for GP Variation : ((x * np.cos(np.cos((np.cos((np.sin(np.sin(x)) + (x / x))) ** 2)))) - ((np.cos(np.cos(np.sin
((x * (np.cos((np.sin(np.sin(x)) + (x / x))) ** 2))))) + np.sin(x)) ** 2))
Best fitness for GP Variation : 1.6485116929439074
```

MSE: 0.6066080

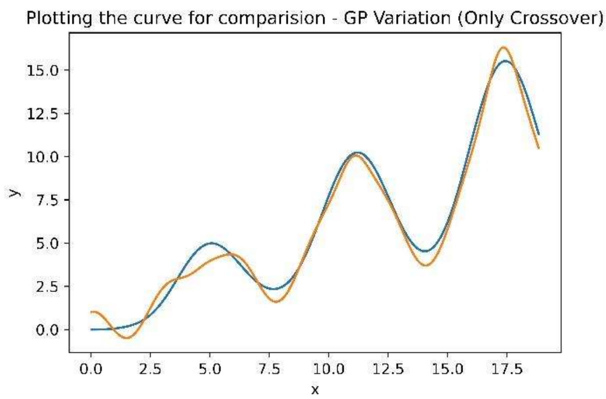

Fig 4: Genetic Algorithm Best Fit Curve Variation (Only Crossover)

DESCRIPTION

1) **Description of representation used**

- The given data set is represented in terms of (x, y).
- For the results we fit the x values in that function to obtain a list of y values.
- We have a mix of variables, constants, and functions. In this case we have the functions addition, subtraction, multiplication, division, sine, cosine, and exponential.
- We upload the function in the form of a Binary heap data structure, and we store it as dictionary. The tree structure is stored in the form of a dictionary in our code.
- Thus, the Genetic Program is run, and it is evaluated from the bottom of the tree, and it moves up the array from the last element to the first element; replacing every "x" with the numerical value where the function is to be evaluated and replacing every operator with the result of that operator applied to its children.

2) **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 and Check if fitness has improved and if so, append.

3) **Hill Climber Algorithm**:

- 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.

4) **Genetic Programming**:

- 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 randomly 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.

5) **Description of EA variation operators used**

In the Genetic Program, we use various terms to represent the code. **Genes** is considered and represented in terms of (x, y). Population is the set of total possible routes. Next, we define the **fitness** which means, the efficiency of the program and this is calculated by using **MSE** (Mean squared error). **Elite Population** is the function that carries the best population to the next step. **Mutation** is used to introduce variation in the population by randomly swapping two points. The set of parents used to create the next set of population is called the **Tournament pool**. To generate set of new values, breeding is done, and this is carried out by a step called **crossover**. In the Genetic program, **pop size** is the size of the defined population. The **rate of mutation** is the rate at which mutation should occur and the **generation** is the total number of iterations.

- **Crossover** is the principal technique of blending genetic material between individuals which is controlled by the crossover parameter. Not like alternative genetic operations, it needs 2 tournaments to be run to seek out a parent and a donor. Crossover takes the winner of a tournament and selects a random subtree from it to be replaced. A second tournament is performed to find a donor. The donor conjointly contains a subtree designated randomly and this can be inserted into the initial parent to make an offspring within the next generation.
- **Subtree mutation** is one in all a lot of aggressive mutation operations and is controlled by the parameter. The rationale it's more aggressive is that more genetic material is replaced by whole naive random components. this could introduce extinct functions and operators into the population to keep up diversity. Subtree mutation takes the winner of a tournament and selects a random subtree from it to be replaced. A donor subtree is generated randomly, and this is often inserted into the parent to make an offspring within the next generation.
- **Point mutation** takes the winner of a tournament and randomly selects nodes to be replaced. Terminals are replaced with other terminals, and functions are replaced with other functions that require the same number of arguments as the original node. The resulting tree forms an offspring into the next generation.

6) Description of EA selection methods used

The Selection methods used is the **Tournament selection method**. 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.

7) Analysis of Performance

What worked?

From our inference, we noticed the increase in the initial population size and the tournament size results in better solutions. When the variations on EA was applied, we get better solutions. Crossover operators help us skip the local maxima/minima. When the number of generations were increased, the fitness is better.

What Didn't work?

In our case, Increasing the initial population size resulted in increasing the computational time and did not add any significant improvement. When the parameters i.e., population size, tournament size and generation weren't set at the right values; the regression curve never happened accurately. When the value of generation was reduced, the paths never converged, and it always plateaued at a particular point.

Learning Curves RS, RMHC, GA and GA Variation:

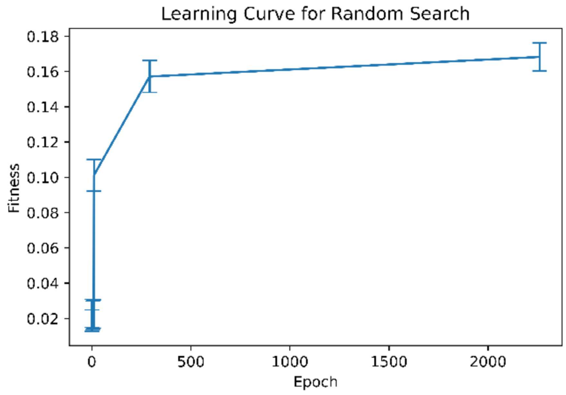

Fig 5: RS Learning Curve

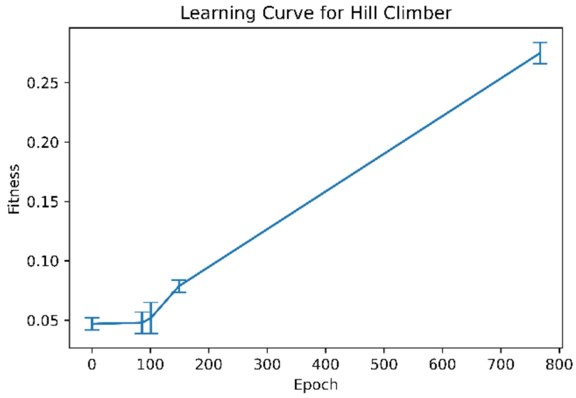

Fig 6: RMHC Learning Curve

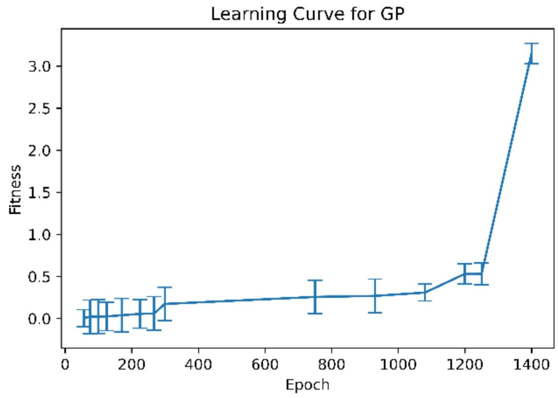

Fig 7: GP Learning Curve

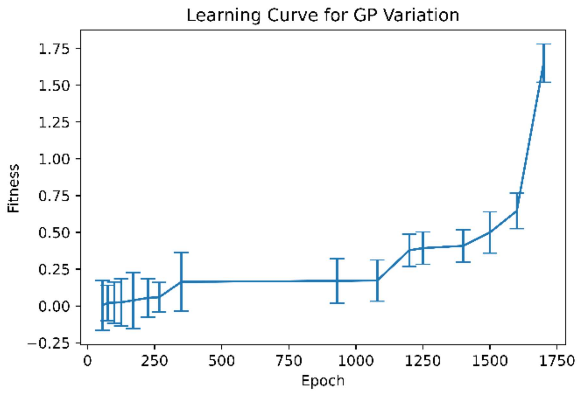

Fig 8: GP Variation Learning Curve

Compiled Learning Curves:

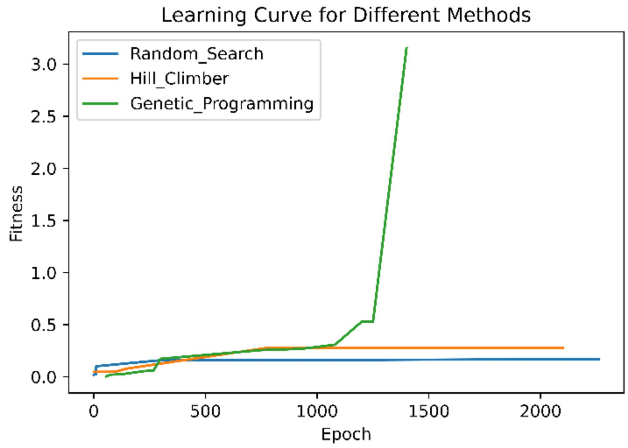

Fig 9: Compiled Learning Curves

DOT Plot Genetic Algorithm:

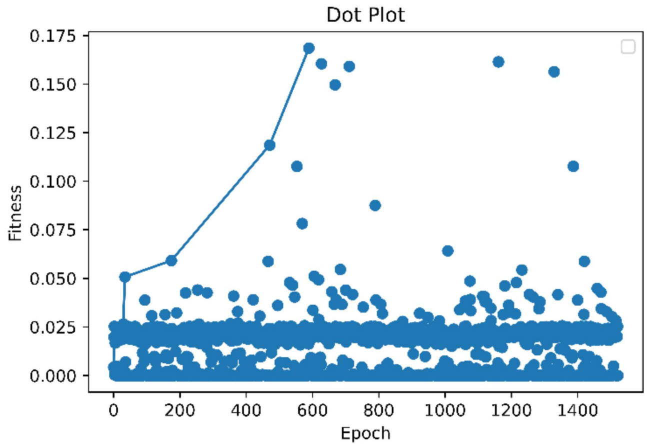

Fig 10: Dot Plot for Genetic Algorithm (Fitness vs Epoch)

Convergence Plot GP:

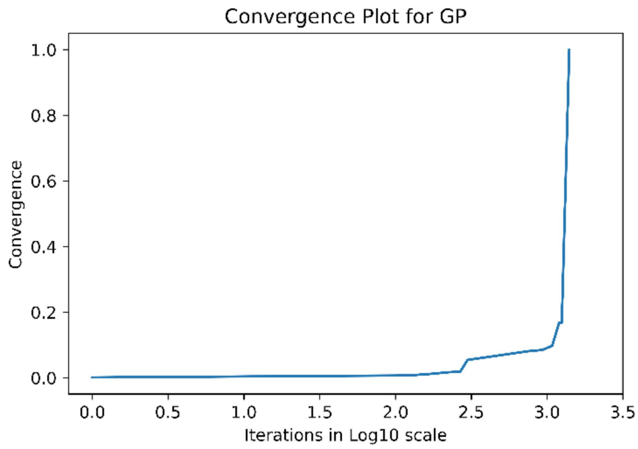

Fig 11: Convergence Plot for GP

Accuracy Curve for GP:

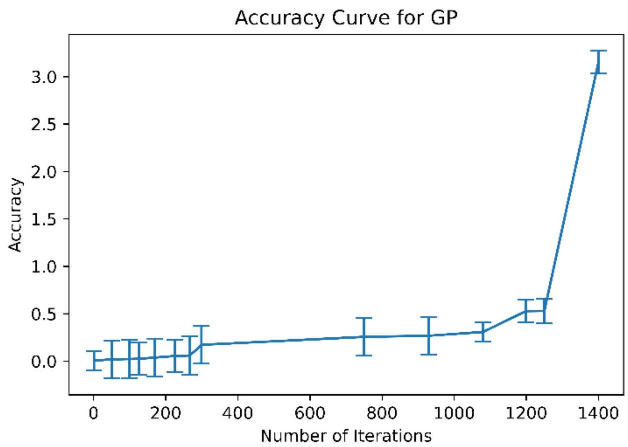

Fig 12: Accuracy Curve for GP

Tree drawn representing best solution for GP:

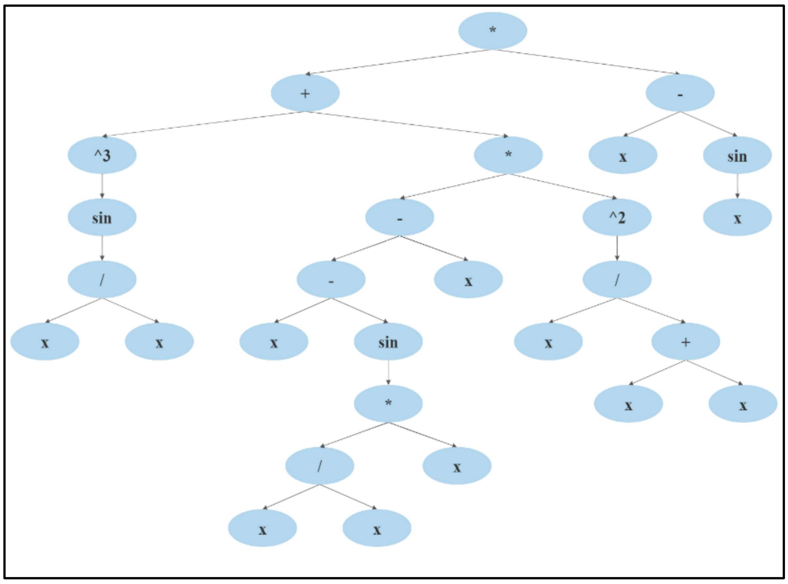

Fig 13: Tree drawn for best solution (GP)

Animation GP (Video Uploaded):

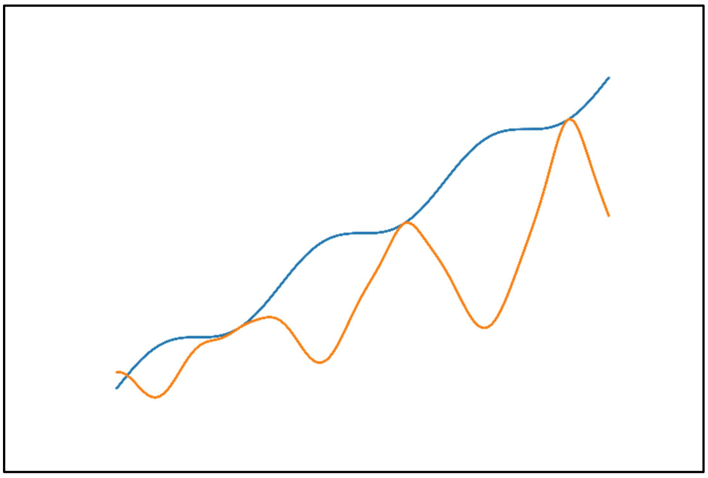

Fig 14: Animation Video for GP

https://drive.google.com/file/d/1CrGAWkeTLDv578qIf8a3hQJQYgoi0yD/view?usp=sharing

(Please access the video using your lion mail)

```python
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)"}
)
```

```python
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)+"\nFitness
:"+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)]
```