```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)]
```