import random

import numpy as np

from sklearn.tree import DecisionTreeRegressor

from sklearn.model\_selection import train\_test\_split

from sklearn.datasets import fetch\_california\_housing

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error

# Step 1: Define the Problem (Optimization task)

# Objective: Minimize Mean Squared Error (MSE) for Decision Tree

def create\_dt\_model(max\_depth, min\_samples\_split, min\_samples\_leaf):

# Create Decision Tree model with hyperparameters

model = DecisionTreeRegressor(max\_depth=max\_depth,

min\_samples\_split=min\_samples\_split,

min\_samples\_leaf=min\_samples\_leaf)

return model

# Step 2: Initialize Parameters for the Genetic Expression Algorithm

POP\_SIZE = 20 # Population size (number of individuals)

NUM\_GENERATIONS = 10 # Number of generations

MUTATION\_RATE = 0.1 # Mutation rate

CROSSOVER\_RATE = 0.7 # Crossover rate

MAX\_DEPTH\_RANGE = (3, 15) # Max depth of decision tree (range)

MIN\_SAMPLES\_SPLIT\_RANGE = (2, 10) # Min samples required for split

MIN\_SAMPLES\_LEAF\_RANGE = (1, 10) # Min samples required for leaf

# Step 3: Initialize Population (Random hyperparameters for decision trees)

class Individual:

def \_\_init\_\_(self):

self.max\_depth = random.randint(MAX\_DEPTH\_RANGE[0], MAX\_DEPTH\_RANGE[1])

self.min\_samples\_split = random.randint(MIN\_SAMPLES\_SPLIT\_RANGE[0], MIN\_SAMPLES\_SPLIT\_RANGE[1])

self.min\_samples\_leaf = random.randint(MIN\_SAMPLES\_LEAF\_RANGE[0], MIN\_SAMPLES\_LEAF\_RANGE[1])

self.fitness = None # To be calculated later

def evaluate(self, X\_train, y\_train, X\_val, y\_val):

# Step 4: Fitness evaluation (train the decision tree and evaluate performance)

model = create\_dt\_model(self.max\_depth, self.min\_samples\_split, self.min\_samples\_leaf)

model.fit(X\_train, y\_train)

# Predict and calculate Mean Squared Error (MSE) on validation data

predictions = model.predict(X\_val)

mse = mean\_squared\_error(y\_val, predictions)

self.fitness = mse

return self.fitness

# Step 5: Selection (Tournament Selection)

def selection(population):

tournament\_size = 3

selected = []

for \_ in range(len(population)):

tournament = random.sample(population, tournament\_size)

tournament.sort(key=lambda x: x.fitness) # Sort by fitness

selected.append(tournament[0]) # Best individual from the tournament

return selected

# Step 6: Crossover (Single-Point Crossover)

def crossover(parent1, parent2):

child1 = Individual()

child2 = Individual()

# Perform crossover on hyperparameters

crossover\_point = random.randint(0, 2)

if crossover\_point == 0:

child1.max\_depth = parent1.max\_depth

child2.max\_depth = parent2.max\_depth

else:

child1.max\_depth = parent2.max\_depth

child2.max\_depth = parent1.max\_depth

crossover\_point = random.randint(0, 2)

if crossover\_point == 0:

child1.min\_samples\_split = parent1.min\_samples\_split

child2.min\_samples\_split = parent2.min\_samples\_split

else:

child1.min\_samples\_split = parent2.min\_samples\_split

child2.min\_samples\_split = parent1.min\_samples\_split

crossover\_point = random.randint(0, 2)

if crossover\_point == 0:

child1.min\_samples\_leaf = parent1.min\_samples\_leaf

child2.min\_samples\_leaf = parent2.min\_samples\_leaf

else:

child1.min\_samples\_leaf = parent2.min\_samples\_leaf

child2.min\_samples\_leaf = parent1.min\_samples\_leaf

return child1, child2

# Step 7: Mutation (Random Mutation)

def mutation(individual):

# Randomly mutate hyperparameters with a certain probability

if random.random() < MUTATION\_RATE:

mutation\_type = random.choice(["max\_depth", "min\_samples\_split", "min\_samples\_leaf"])

if mutation\_type == "max\_depth":

individual.max\_depth = random.randint(MAX\_DEPTH\_RANGE[0], MAX\_DEPTH\_RANGE[1])

elif mutation\_type == "min\_samples\_split":

individual.min\_samples\_split = random.randint(MIN\_SAMPLES\_SPLIT\_RANGE[0], MIN\_SAMPLES\_SPLIT\_RANGE[1])

elif mutation\_type == "min\_samples\_leaf":

individual.min\_samples\_leaf = random.randint(MIN\_SAMPLES\_LEAF\_RANGE[0], MIN\_SAMPLES\_LEAF\_RANGE[1])

# Step 8: Gene Expression (Gene representation of decision tree hyperparameters)

# Step 9: Iterate over generations (Evolution process)

def evolve\_population(X\_train, y\_train, X\_val, y\_val):

# Step 1: Initialize population with random decision tree configurations

population = [Individual() for \_ in range(POP\_SIZE)]

best\_solution = None

best\_fitness = float('inf')

for generation in range(NUM\_GENERATIONS):

# Step 2: Evaluate the population

for individual in population:

individual.evaluate(X\_train, y\_train, X\_val, y\_val)

# Track best solution

current\_best = min(population, key=lambda x: x.fitness)

if current\_best.fitness < best\_fitness:

best\_fitness = current\_best.fitness

best\_solution = current\_best

# Step 3: Selection (Tournament selection)

selected = selection(population)

# Step 4: Crossover (Single-point crossover)

next\_generation = []

for i in range(0, len(selected), 2):

parent1, parent2 = selected[i], selected[i + 1]

if random.random() < CROSSOVER\_RATE:

child1, child2 = crossover(parent1, parent2)

next\_generation.extend([child1, child2])

else:

next\_generation.extend([parent1, parent2])

# Step 5: Mutation

for individual in next\_generation:

mutation(individual)

# Replace old population with new generation

population[:] = next\_generation

# Output the best solution so far

print(f"Generation {generation + 1}, Best Fitness (MSE): {best\_fitness}")

return best\_solution

# Load and prepare the dataset (California Housing Dataset for regression task)

data = fetch\_california\_housing()

X = data.data

y = data.target

# Normalize the dataset

scaler = StandardScaler()

X = scaler.fit\_transform(X)

# Split dataset into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Run the GEA to optimize the decision tree configuration

best\_solution = evolve\_population(X\_train, y\_train, X\_val, y\_val)

# Output the best solution found

print("Best Decision Tree Configuration:")

print(f"Max Depth: {best\_solution.max\_depth}")

print(f"Min Samples Split: {best\_solution.min\_samples\_split}")

print(f"Min Samples Leaf: {best\_solution.min\_samples\_leaf}")

output:

Generation 1, Best Fitness (MSE): 0.3620545146330497

Generation 2, Best Fitness (MSE): 0.3614904659321508

Generation 3, Best Fitness (MSE): 0.3614751198592556

Generation 4, Best Fitness (MSE): 0.3614751198592556

Generation 5, Best Fitness (MSE): 0.36147334957668786

Generation 6, Best Fitness (MSE): 0.3614731899752306

Generation 7, Best Fitness (MSE): 0.3614731899752306

Generation 8, Best Fitness (MSE): 0.3614731899752306

Generation 9, Best Fitness (MSE): 0.3614731899752306

Generation 10, Best Fitness (MSE): 0.3614731899752306

Best Decision Tree Configuration:

Max Depth: 12

Min Samples Split: 3

Min Samples Leaf: 9