```
%%writefile evaluate_fitness.py
import numpy as np
import pandas as pd
import sys
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score



chromosome_id = int(sys.argv[1])

# Load dataset (do not drop any columns)
df = pd.read_csv("Crop_recommendation.csv")
X = df.drop(columns="label")  # Ensure all features are included
y = df["label"]

# Read chromosome
chromosome = np.loadtxt("chromosome.txt", delimiter=",", dtype=int)
print(f"Chromosome {chromosome_id}:", chromosome)

# Validate chromosome length
if len(chromosome) != X.shape[1]:
    raise ValueError(f"Chromosome length {len(chromosome)} does not match the number of features {X.shape[1]}")

# Get selected features
selected_features = np.where(chromosome == 1)[0]
print("Selected Features:", selected_features)
print("Dataset Shape:", X.shape)

# Handle case where no features are selected
if len(selected_features) == 0:
    print("No features selected. Exiting...")
    with open("fitness_result.txt", "w") as f:
        f.write("0.0")
    exit()

# Select features based on chromosome
X_selected = X.iloc[:, selected_features]

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.3, random_state=42)

# Train the model
rf = RandomForestClassifier(n_estimators=50, random_state=42, criterion="entropy")
rf.fit(X_train, y_train)

# Predict and calculate accuracy
y_pred = rf.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Save fitness result
with open("fitness_result.txt", "w") as f:
    f.write(str(accuracy))
```

```
Overwriting evaluate_fitness.py
```

```
%%writefile main.c



#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define POP_SIZE 100
#define CHROMOSOME_LENGTH 7
#define NUM_GENERATIONS 50
#define MUTATION_RATE 0.01
#define MAX_STAGNATION 3 // Maximum allowed generations without improvement
```

```
#define MAX_STAGNATION 5 // Maximum allowed generations without improvement

// Individual struct to include ID
typedef struct {
    int id;                     // ID of the individual
    int genes[CHROMOSOME_LENGTH]; // Chromosome (gene array)
    double fitness;             // Fitness value
} Individual;

// Function prototypes
void generate_chromosome(int chromosome[], int length);
double evaluate_fitness(int chromosome[], int length, int chromosome_id);
int roulette_wheel_selection(Individual population[], int population_size);
void three_point_crossover(int parent1[], int parent2[], int offspring1[], int offspring2[], int length);
void mutate(int chromosome[], int length, double mutation_rate);

int main() {
    srand(time(NULL));

    Individual population[POP_SIZE];

    // Step 1: Initialize Population (silent initialization)
    for (int i = 0; i < POP_SIZE; i++) {
        population[i].id = i + 1; // Assign unique ID
        generate_chromosome(population[i].genes, CHROMOSOME_LENGTH);
        population[i].fitness = evaluate_fitness(population[i].genes, CHROMOSOME_LENGTH, population[i].id);
    }

    double best_fitness = population[0].fitness;
    int stagnation_count = 0;

    // Genetic Algorithm Loop
    for (int generation = 0; generation < NUM_GENERATIONS; generation++) {
        printf("\n=== Generation %d ===\n", generation);

        Individual new_population[POP_SIZE];

        // Step 1: Find the Best Individual
        double current_best_fitness = population[0].fitness;
        int best_idx = 0;
        for (int i = 1; i < POP_SIZE; i++) {
            if (population[i].fitness > current_best_fitness) {
                current_best_fitness = population[i].fitness;
                best_idx = i;
            }
        }

        // Step 2: Add the Best Individual to the New Population (Elitism)
        new_population[0] = population[best_idx];

        // Step 3: Generate the Rest of the New Population
        for (int i = 1; i < POP_SIZE; i += 2) { // Start from 1 to skip the best individual
            // Selection
            int parent1_idx = roulette_wheel_selection(population, POP_SIZE);
            int parent2_idx = roulette_wheel_selection(population, POP_SIZE);

            // Crossover
            three_point_crossover(
                population[parent1_idx].genes, population[parent2_idx].genes,
                new_population[i].genes, new_population[i + 1].genes, CHROMOSOME_LENGTH
            );

            // Mutation
            mutate(new_population[i].genes, CHROMOSOME_LENGTH, MUTATION_RATE);
            if (i + 1 < POP_SIZE) { // Avoid out-of-bounds
                mutate(new_population[i + 1].genes, CHROMOSOME_LENGTH, MUTATION_RATE);
            }

            // Evaluate fitness of offspring
            new_population[i].id = i + 1;
            if (i + 1 < POP_SIZE) {
                new_population[i + 1].id = i + 2;
                new_population[i + 1].fitness = evaluate_fitness(new_population[i + 1].genes, CHROMOSOME_LENGTH, new_population[i + 1].id);
            }
            new_population[i].fitness = evaluate_fitness(new_population[i].genes, CHROMOSOME_LENGTH, new_population[i].id);
        }

        // Replace the old population with the new population
        for (int i = 0; i < POP_SIZE; i++) {
            population[i] = new_population[i];
        }

        // Log population for the current generation
        for (int i = 0; i < POP_SIZE; i++) {
            printf("Chromosome %d: [", population[i].id);
            for (int j = 0; j < CHROMOSOME_LENGTH; j++) {
                printf("%d", population[i].genes[j]);
                if (j < CHROMOSOME_LENGTH - 1) printf(" ");
            }
            printf("] Accuracy: %.4f\n", population[i].fitness);
        }

        printf("Best Fitness in Generation %d: %.4f (ID: %d)\n", generation, current_best_fitness, population[best_idx].id);

        // Step 4: Check for Stagnation
        if (current_best_fitness == best_fitness) {
            stagnation_count++;
            printf("No improvement in generation %d. Stagnation count: %d\n", generation, stagnation_count);
            if (stagnation_count >= MAX_STAGNATION) {
                printf("Stopping early due to stagnation: No improvement for %d generations.\n", MAX_STAGNATION);
                break;
            }
        } else {
            best_fitness = current_best_fitness;
            stagnation_count = 0;
        }
    }

    return 0;
}

void generate_chromosome(int chromosome[], int length) {
    for (int i = 0; i < length; i++) {
        chromosome[i] = rand() % 2; // Randomly assign 0 or 1
    }
}

double evaluate_fitness(int chromosome[], int length, int chromosome_id) {
    FILE *file = fopen("chromosome.txt", "w");
    if (!file) {
        perror("Error opening chromosome file");
        exit(EXIT_FAILURE);
    }

    // Write chromosome to file
    for (int i = 0; i < length; i++) {
        fprintf(file, "%d", chromosome[i]);
        if (i < length - 1) {
            fprintf(file, ",");
        }
    }
    fclose(file);

    // Pass chromosome ID to Python script
    char command[256];
    sprintf(command, "python3 evaluate_fitness.py %d", chromosome_id);
    if (system(command) != 0) {
        fprintf(stderr, "Error executing Python script.\n");
        exit(EXIT_FAILURE);
    }

    // Read fitness result
    file = fopen("fitness_result.txt", "r");
    if (!file) {
        perror("Error opening fitness result file");
        exit(EXIT_FAILURE);
    }
    double fitness;
    fscanf(file, "%lf", &fitness);
    fclose(file);

    return fitness;
}

int roulette_wheel_selection(Individual population[], int population_size) {
    double total_fitness = 0.0;
    for (int i = 0; i < population_size; i++) {
        total_fitness += population[i].fitness;
    }

    double cumulative_prob[POP_SIZE];
    cumulative_prob[0] = population[0].fitness / total_fitness;
    for (int i = 1; i < population_size; i++) {
        cumulative_prob[i] = cumulative_prob[i - 1] + population[i].fitness / total_fitness;
    }

    double r = (double)rand() / RAND_MAX;
    for (int i = 0; i < population_size; i++) {
        if (r <= cumulative_prob[i]) {
            return i;
        }
    }
    return population_size - 1;
}

void three_point_crossover(int parent1[], int parent2[], int offspring1[], int offspring2[], int length) {
    int swap_start = length - 3; // Start of the last 3 features

    for (int i = 0; i < length; i++) {
        if (i < swap_start) {
            offspring1[i] = parent1[i];
            offspring2[i] = parent2[i];
        } else {
            offspring1[i] = parent2[i];
            offspring2[i] = parent1[i];
        }
    }
}

void mutate(int chromosome[], int length, double mutation_rate) {
    for (int i = 0; i < length; i++) {
        double r = (double)rand() / RAND_MAX;
        if (r < mutation_rate) {
            chromosome[i] = 1 - chromosome[i]; // Flip bit
        }
    }
}
```

```
Overwriting main.c
```

Start coding or generate with AI.

```
!gcc -o main main.c

import time

t = time.time()
!./main
```

```
Chromosome 70: [1 0 0 0 0 1 1] Accuracy: 0.7530
Chromosome 71: [0 0 1 0 1 1 1] Accuracy: 0.9727
Chromosome 72: [1 1 1 0 0 1 1] Accuracy: 0.9636
Chromosome 73: [1 0 0 0 1 0 0] Accuracy: 0.6379
Chromosome 74: [0 1 0 1 0 1 1] Accuracy: 0.9091
Chromosome 75: [0 0 1 0 1 1 1] Accuracy: 0.9727
Chromosome 76: [0 1 1 0 0 1 1] Accuracy: 0.9091
Chromosome 77: [0 1 1 0 0 1 1] Accuracy: 0.9091
Chromosome 78: [0 1 1 0 0 1 1] Accuracy: 0.9091
Chromosome 79: [1 1 0 1 1 1 0] Accuracy: 0.9364
Chromosome 80: [0 1 1 0 1 1 1] Accuracy: 0.9788
Chromosome 81: [0 0 1 0 1 0 0] Accuracy: 0.7061
Chromosome 82: [0 0 1 1 0 1 0] Accuracy: 0.6833
Chromosome 83: [0 0 1 1 1 0 1] Accuracy: 0.9742
Chromosome 84: [0 0 1 1 0 1 1] Accuracy: 0.8864
Chromosome 85: [1 0 0 1 0 1 1] Accuracy: 0.8803
Chromosome 86: [0 1 1 0 1 0 1] Accuracy: 0.9682
Chromosome 87: [0 0 1 1 1 0 0] Accuracy: 0.8485
Chromosome 88: [1 1 0 1 0 1 1] Accuracy: 0.9515
Chromosome 89: [1 0 0 1 1 1 1] Accuracy: 0.9530
Chromosome 90: [0 1 1 0 1 0 0] Accuracy: 0.8697
Chromosome 91: [0 1 1 0 1 0 1] Accuracy: 0.9682
Chromosome 92: [1 0 1 0 1 0 1] Accuracy: 0.9727
Chromosome 93: [0 1 0 0 1 1 0] Accuracy: 0.7455
Chromosome 94: [1 0 0 0 1 1 0] Accuracy: 0.7333
Chromosome 95: [1 1 1 0 0 1 1] Accuracy: 0.9636
Chromosome 96: [1 0 0 1 1 0 0] Accuracy: 0.7864
Chromosome 97: [0 1 1 0 0 1 1] Accuracy: 0.9091
Chromosome 98: [1 1 1 1 1 0 1] Accuracy: 0.9924
Chromosome 99: [1 1 0 1 0 0 1] Accuracy: 0.9273
Chromosome 100: [0 0 1 0 1 0 1] Accuracy: 0.9455
Best Fitness in Generation 3: 0.9924 (ID: 92)
No improvement in generation 3. Stagnation count: 3
Stopping early due to stagnation: No improvement for 3 generations.
*** stack smashing detected ***: terminated
```

```
delta_t = time.time()-t
print("Time taken for total execution of serial code:", delta_t,"seconds")
serial_time = delta_t
```

```
Time taken for total execution of serial code: 903.0316009521484 seconds
```