ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

д-р техн. наук, профессор		Скобцов Ю.А.
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ

Эволюционные алгоритмы оценки стоимости проектов в программной инженерии

по курсу: Эволюционные методы проектирования программно-информационных систем

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4132		Н.И. Карпов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2024

СОДЕРЖАНИЕ

1 Индивидуальное задание

В данной лабораторной работе необходимо определить коэффициенты a и b для модели COCOMO с использованием генетического алгоритма.

Используемое кодирование решения – вещественное;

Оператор мутации – арифметический. Оператор кроссинговера – арифметический. Оператор репродукции – рулетка.

Метод отбора обучающего и тестового множества из входных данных – случайный с перемешиванием.

Функция ошибки – евклидово расстояние (ED).

2 Краткие теоретические сведения

Одной из самых популярных моделей, используемых для оценки сложности проектируемого программного обеспечения (ПО), является модель COCOMO (COnstructive COst Model). Эта модель разработана на основе фактически статистики 63 проектов ПО (НАСА). Модель позволяет определить математическую зависимость между сложностью ПО, выраженную в килостроках кода, и затратами на его разработку, которые оцениваются в человеко-месяцах. Ядром модели является следующая формула Ef=aL^b , где L - длина кода ПО в килостроках; Ef – оценка сложности проекта в человеко-месяцах; a и b - коэффициенты (параметры) модели, которые для различных типов ПО имеют различные значения.

Основная проблема модель COCOMO заключается в том, что она не обеспечивает реальных оценок на затраты при проектировании ПО в современных условиях. Т.е. оценка программного обеспечения на основе существующих параметров не всегда дает точный результат; из-за этого часто требуется настройка параметров для получения более точных результатов. Поэтому в настоящее время идет активный поиск новых моделей (или развития и модификаций существующих). Это ограничение модели COCOMO можно преодолеть путем применения методов искусственного интеллекта, таких как искусственные нейронные сети, генетические алгоритмы и другие метаэвристики.

В качестве обучающих данных был выбран набор с расширенным множеством проектов НАСА:

L	Ef	COCOMO	ANN	GA	Hybrid
2.2	8.4	24.15	13.65	8.95	6.32
3.5	10.8	3.95	5.26	4.69	1.13

5.5	18	7.36	5.21	6.75	4.35
6	24	58.88	34.10	27.63	28.02
9.7	25.2	20.05	11.50	13.49	7.61
7.7	31.2	23.91	12.35	7.54	12.42
11.3	36	30.83	17.45	12.45	13.35
8.2	36	29.55	16.68	14.23	11.21
6.5	42	28.32	18.52	11.64	13.42
8	42	22.22	13.21	15.47	9.34
20	48	27.21	14.65	16.32	12.16
10	48	41.66	23.98	19.84	19.84
15	48	46.19	28.04	23.11	26.74
10.4	50	34.90	25.47	17.02	21.95
13	60	9.36	6.53	5.31	7.15
14	60	25.88	15.41	17.54	8.46
19.7	60	6.10	7.21	4.21	2.54
32.5	60	93.91	47.35	56.47	36.10
31.5	60	3.81	6.52	5.46	1.07
12.5	62	27.96	13.11	10.84	4.31
15.4	70	22.51	10.13	12.76	7.02
20	72	60.76	45.68	33.82	27.11
7.5	72	41.75	32.61	24.15	15.04
16.3	82	29.79	23.40	17.37	7.46
15	90	39.54	27.68	21.51	19.01
11.4	98.8	42.04	25.10	19.07	21.74
21	107	36.75	24.55	16.53	9.02
16	114	34.48	24.55	16.53	9.92
25.9	117.6	27.85	19.36	11.57	17.09
24.6	117.6	31.65	21.87	16.34	14.82
29.5	120	18.94	11.15	7.13	6.44
19.3	155	35.78	17.30	21.06	16.72
32.6	170	29.88	19.54	15.19	5.68
35.5	192	32.10	16.35	8.37	13.06
38	210	28.46	13.19	19.50	15.43
48.5	239	24.31	8.43	12.07	7.94
47.5	252	37.81	21.36	18.64	11.83
70	278	21.28	9.42	11.46	6.24
66.6	300	23.76	11.30	16.79	9.22
66.6	352.8	35.17	19.25	11.20	13.62
50	370	36.90	23.54	13.48	7.42
79	400	45.74	31.29	22.97	18.06
90	450	38.29	20.11	31.73	15.94
78	571.4	24.50	13.64	8.03	5.21
100	215	120.66	86.14	61.42	51.04
150	324	49.50	26.80	13.09	23.83
100	360	44.97	17.67	25.07	12.62

100	360	15.85	6.23	8.62	9.84
190	420	1.89	4.87	3.84	2.65
115.8	480	11.37	16.49	5.32	5.42
101	750	19.87	10.67	6.46	12.71
161.1	815	4.76	10.25	8.41	5.95
284.7	973	38.36	21.43	17.09	10.14
227	1181	3.93	2.36	6.31	4.62
177.9	1228	3.64	9.84	5.08	2.06
282.1	1368	17.21	9.46	11.36	7.92
219	2120	29.00	21.03	15.81	8.31
423	2300	25.78	16.07	7.44	9.02
302	2400	0.46	3.24	5.64	2.54
370	3240	25.21	8.62	3.21	6.87

3 Результаты выполнения работы

В программе был реализован пользовательский интерфейс для ввода основных параметров генетического алгоритма:

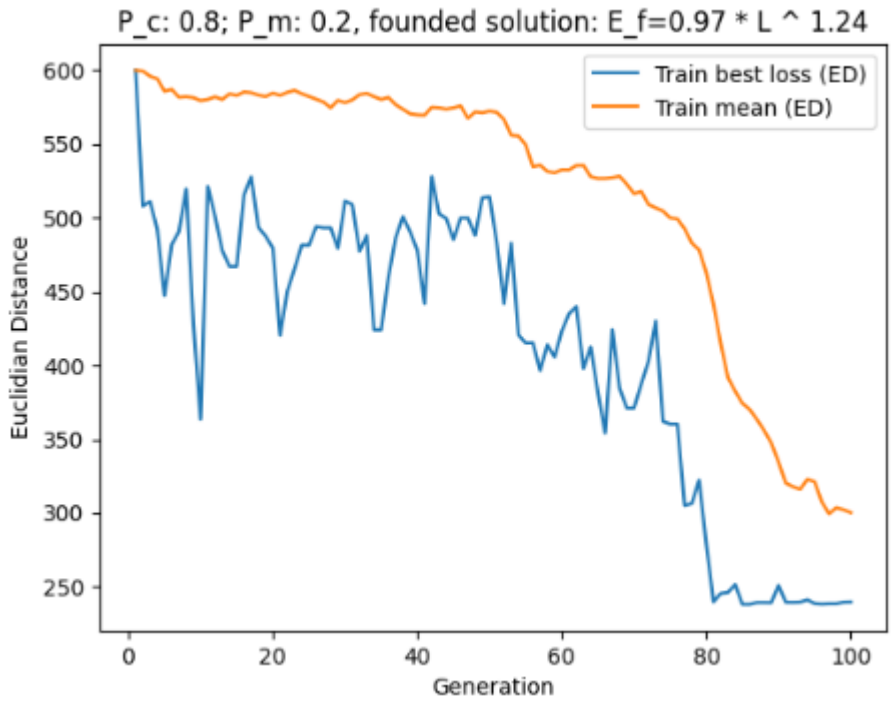
```
Load data and split it into train and test with 80/20
Loading completed!

Enter max generations (default: 100): 100
Enter population size (default: 200): 200
Enter crossover probability (default: 0.8): 0.8
Enter mutation probability (default: 0.2): 0.2

Input completed. Start algorithm...

Generation [1/100]              Min ED = 600.041
```

Результат обучения при введенных параметрах:




```
Generation [99/100]             Min ED = 248.849
Generation [100/100]                Min ED = 250.836

 ================================================================================================
Required time: 0.92s. Result min ED: 250.836193.  Required generations: 100. Pc = 0.8,  Pm = 0.2
```

Ошибка стремится к меньшим значениям, но ввиду «переобучения», ошибка на тестовых данных остаётся довольно большой:

```
Fitting completed!
Start evaluate best solution.
    ED: 473.92

Process finished with exit code 0
```

Лучшим же решением является: $E_f = 0.97\ L^{1.24}$.

4 Листинг программы

```
import pandas as pd
import numpy as np
import genalg

from sklearn.model_selection import train_test_split

TEST_SIZE = 0.2

print(f"Load data and split it into train and test with {100 -
int(TEST_SIZE * 100)}/{int(TEST_SIZE * 100)}")
```

```
df = pd.read_csv("nasa_data.csv")
L = np.array(df["L"])
E = np.array(df["Ef"])

x_train, x_test, y_train, y_test = train_test_split(L, E,
test_size=TEST_SIZE, random_state=1337)

print("Loading completed!\n")


def get_user_input(prompt, default_value, cast_func):
    user_input = input(f"{prompt} (default: {default_value}): ")
    if user_input.strip() == "":
        return default_value
    try:
        return cast_func(user_input)
    except ValueError:
        print("Invalid input. Using default value.")
        return default_value


MAX_GENERATIONS = get_user_input("Enter max generations", 100, int)
POPULATION_SIZE = get_user_input("Enter population size", 200, int)
CROSSOVER_PROBABILITY = get_user_input("Enter crossover probability", 0.8,
float)
MUTATION_PROBABILITY = get_user_input("Enter mutation probability", 0.2,
float)

print("\nInput completed. Start algorithm...\n")

genetic_optimizer = genalg.RegressionGeneticAlgorithm(POPULATION_SIZE,
MAX_GENERATIONS, CROSSOVER_PROBABILITY,
                                                      MUTATION_PROBABILITY)
genetic_optimizer.fit(x_train, y_train)

print("\nFitting completed!")
genetic_optimizer.evaluate_best_solution(x_test, y_test)

import matplotlib.pyplot as plt
import numpy as np

from time import time

import random


def cocomo_count(c, code_len):
    return c[0] * code_len ** c[1]


class RegressionGeneticAlgorithm:
    """
    Solves task of finding maximum of target function (fitness function)
using genetic algorithm.
    :param population_size: size of the population
    :param max_generations: number of max available generations. For a sit-
uation when it is impossible to find a satisfying optimum
    :param crossover_probability: probability of a crossover
    :param mutation_probability: probability of a mutation
    """
```

```
    def __init__(self, population_size: int, max_generations: int, crosso-
ver_probability: float, mutation_probability: float):
        self.population_size = population_size
        self.max_generations = max_generations
        self.crossover_probability = crossover_probability
        self.mutation_probability = mutation_probability

        self.population = np.ones((population_size, 2))

        self.losses_values = np.ones(population_size, float)
        self.mean_errors = []
        self.min_errors = []

    def fit(self, x, y_true):
        self.x = x
        self.y_true = y_true

        generation = 1  # generations counter

        start_time = time()

        while generation <= self.max_generations:
            # 1. Evaluate fitness.
            self.evaluate()

            # 2. Reproduction.
            self.reproduction()

            # 3. Crossing.
            self.crossover()

            # 4. Mutate.
            self.mutate()

            print(f'Generation [{generation}/{self.max_generations}]\t\t' +
                  f'\t\t Min ED = {min(self.losses_values):.3f}\t\t')

            generation += 1

        end_time = time()

        best_solution_idx = np.where(self.losses_values ==
min(self.losses_values))[0][0]
        self.best_solution = self.population[best_solution_idx]

        print('\n', '=' * 100)
        print(f'Required time: {end_time - start_time:.2f}s. Result min ED:
{min(self.losses_values):4f}. ',
              f'Required generations: {generation - 1}. Pc =
{self.crossover_probability}, ',
              f'Pm = {self.mutation_probability}\n',
              f'Found solution: Ef={self.best_solution[0]:.2f} * L ^
{self.best_solution[1]:.2f}')

        self.draw_errors()

    def evaluate_best_solution(self, x_test, y_test):
        print("Start evaluate best solution.")

        y = cocomo_count(self.best_solution, x_test)
```

```
        ed = np.sqrt(np.mean(np.power(y_test - y, 2)))

        print("\tED: {:.2f}".format(ed))

    def evaluate(self):
        """
        Evaluates the fitness of the population.
        """
        # Count fitness of every chromosome in population.
        for i in range(self.population_size):
            y = cocomo_count(self.population[i], self.x)
            self.losses_values[i] = np.sqrt(np.mean(np.power(self.y_true -
y, 2)))

        self.mean_errors.append(np.mean(self.losses_values))
        self.min_errors.append(np.min(self.losses_values))

    def reproduction(self):
        """
        Roulette based reproduction algorithm.
        """
        fitness_copy = self.losses_values.copy()
        fitness_copy_inverted = 1 / fitness_copy

        total_fitness = sum(fitness_copy_inverted)
        probabilities = [fitness / total_fitness for fitness in fit-
ness_copy_inverted]
        new_population = []

        for i in range(self.population_size):
            current_wheel_probability = 0.
            random_probability = random.random()

            # Emulate roulette pass.
            for j in range(self.population_size):
                current_wheel_probability += probabilities[j]

                if random_probability < current_wheel_probability:
                    # Random chosen probability in range
[last_chromosome_p, new_chromosome_p).
                    new_population.append(self.population[j])
                    break

            # If there is case with rounding error (last chromosome wasn't
added) add last chromosome manually.
            if i != len(new_population) - 1:
                new_population.append(self.population[-1])

        # Update population.
        for i in range(self.population_size):
            self.population[i] = new_population[i]

    def crossover(self):
        """
        Randomly distributes the population into pairs and applies arithme-
tic crossover with a given probability.
        """
        random_indexes = np.random.permutation(self.population_size)

        for i in range(0, len(random_indexes) - 1, 2):
            if np.random.rand() < self.crossover_probability:
```

```
                # Select parents
                parent1 = self.population[random_indexes[i]]
                parent2 = self.population[random_indexes[i + 1]]

                # Generate offspring using arithmetic crossover
                alpha = np.random.rand()  # Weight for linear combination
                offspring1 = alpha * parent1 + (1 - alpha) * parent2
                offspring2 = alpha * parent2 + (1 - alpha) * parent1

                # Replace parents with offspring
                self.population[random_indexes[i]] = offspring1
                self.population[random_indexes[i + 1]] = offspring2

    def mutate(self):
        """
        Applies arithmetic mutation to the population with a given proba-
bility.
        """
        for i in range(self.population_size):
            if np.random.rand() < self.mutation_probability:
                # Select a chromosome to mutate
                chromosome = self.population[i]

                # Apply arithmetic mutation
                alpha = np.random.rand()  # Weight for mutation
                mutation_vector = np.random.uniform(-0.1, 0.1,
size=chromosome.shape)  # Small random changes
                self.population[i] = (1 - alpha) * chromosome + alpha *
(chromosome + mutation_vector)

    def draw_errors(self):
        figure = plt.subplot()
        iters = np.array(range(1, self.max_generations + 1))
        plt.plot(iters, self.min_errors, label=fr"Train best loss (ED)")
        plt.plot(iters, self.mean_errors, label=fr"Train mean (ED)")
        plt.xlabel("Generation")
        plt.ylabel("Euclidian Distance")

        plt.title(
            fr"P_c: {self.crossover_probability}; P_m:
{self.mutation_probability}, founded solution:
E_f={self.best_solution[0]:.2f} * L ^ {self.best_solution[1]:.2f}")
        plt.legend()

        plt.show()
```

5 Вывод

В ходе работы была создана программа для решения задачи поиска коэффициентов модели COCOMO с использованием генетического алгоритма. Полученное решение быстро сходится к неплохой ошибке. Однако, ошибка на тестовых данных, немного уменьшаясь, остаётся большой.

Таким образом, рассмотренный способ улучшения модели COCOMO является неплохой альтернативой другим способам оценки стоимости программных проектов.