Projekt 3 – Wykorzystanie bibliotek PyGAD do algorytmów genetycznych w języku Python

Grupa 3 osobowa

[Projekt][OE][DataScience][24/25]

1 Cel projektu

Optymalizacja funkcji z wykorzystaniem biblioteki PyGAD w języku Python.

2 Założenia i implementacja

1. Wykorzystanie szkieletu projektu z `example_02.py`.
2. Zakres parametrów w przedziale [0,1], konfiguracja:
   - `init_range_low = 0`
   - `init_range_high = 2`
   - `gene_type = int`
3. Dekodowanie osobnika (np. 60 bitów = 3 zmienne po 20 bitów).
4. Funkcja celu (uwaga decodeInd to pseudo kod, trzeba napisać własną metodę) Funkcja celu to wybrana przez nas w projekcie P1 i/lub P2:

```
def fitnessFunction(individual):
    ind = decodeInd(individual)
    result = (ind[0] + 2 * ind[1] - 7)**2 + (2 * ind[0] + ind[1] - 5)**2
    return result
```

5. Testowane metody selekcji:
   - turniejowa (tournament)
   - koło ruletki (rws)
   - losowa (random)
6. Testowane metody krzyżowania:
   - jednopunktowe (single_point)
   - dwupunktowe (two_points)
   - jednorodne (uniform)

7. Przykład własnej funkcji krzyżowania:

```
def crossover_func(parents, offspring_size, ga_instance):
    offspring = []
    idx = 0
    while len(offspring) != offspring_size[0]:
        parent1 = parents[idx % parents.shape[0], :].copy()
        parent2 = parents[(idx + 1) % parents.shape[0], :].copy()
        random_split_point = numpy.random.choice(range(offspring_size[1]))
        parent1[random_split_point:] = parent2[random_split_point:]
        offspring.append(parent1)
        idx += 1
    return numpy.array(offspring)
```

8. Testowane metody mutacji:
   - losowa (random)
   - zamiana indeksów (swap)
9. Konfiguracja klasy `GA()` z PyGAD:

```
ga_instance = pygad.GA(
    num_generations=num_generations,
    sol_per_pop=sol_per_pop,
    num_parents_mating=num_parents_mating,
    num_genes=num_genes,
    fitness_func=fitness_func,
    init_range_low=0,
    init_range_high=2,
    gene_type=int,
    mutation_num_genes=mutation_num_genes,
    parent_selection_type=parent_selection_type,
    crossover_type=crossover_type,
    mutation_type=mutation_type,
    keep_elitism=1,
    K_tournament=3,
    random_mutation_max_val=32.768,
    random_mutation_min_val=-32.768,
    logger=logger,
    on_generation=on_generation,
    parallel_processing=['thread', 4]
)
```

10. Alternatywny wariant z reprezentacją rzeczywistą (`gene_type = float`).
11. Implementacja mutacji Gaussa:

```
def mutation_func(offspring, ga_instance):
    for chromosome_idx in range(offspring.shape[0]):
        random_gene_idx = numpy.random.choice(range(offspring.shape[1]))
```

```
        offspring[chromosome_idx, random_gene_idx] += numpy.random.random()
    return offspring
```

3 To do

1. Implementacja w kodzie funkcji nad którą pracowaliśmy w projekcie P1 lub P2, nie trzeba jej kopiować - można ją wywołać z benchmarka. Link do tej biblioteki był w Projekcie 1 w punkcie na temat funkcji testowych:

```
import benchmark_functions as bf
func = bf.Hyperellipsoid(n_dimensions=10)
print(func.suggested_bounds())
print(func.minimum())
```

2. Przetestowanie metod selekcji, krzyżowania, mutacji z punktu "Założenia i implementacja" podpunkty 5, 6 i 8 w tym pliku PDF.

3. Wykonanie testów - obliczeń dla reprezentacji binarnej jak i rzeczywistej.

4. Dodanie przedstawiania wyników: wartości funkcji celu, średniej, odchylenia standardowego na wykresie.

5. Przygotowanie sprawozdania porównującego konfiguracje algorytmu genetycznego dla obu reprezentacji.

4 To do

1. Po wykonaniu zadania przesłać sprawozdanie i kod poprzez Teams lub maila.

2. Obrona projektu: **12.05.2025**