Computational Intelligence: Course Report

Luciana Colella, s328092

February 10, 2025

1 Project: Symbolic Regression

1.1 Author's note

All the implementations of this project have been develop by me, giving my personal understanding and approach to the task. For additional details on the project implementation, see CI2024_project-work Repository

1.2 Overview

The main focus of this project is to solve several symbolic regression problems using genetic programming (GP). The goal is to find different mathematical expressions that approximates given datasets. These datasets consists in an input features (X), with its several variables (e.g., ['x0', 'x1'] and corresponding target values (y). The genetic algorithm evolves a population of mathematical expressions, designed as trees to minimize the error between the predicted and actual target values.

1.3 Data overview

The project has multiple datasets to work on, each with a particular complexity. Each dataset contains: X: Input features (from 1 to 6 variables) with different amount of samples. y: Target output values for each dataset. Objective: Find the most accurate mathematical formula that maps X to y. Dataset details are at Table 1.1.

1.4 Approach

The code implements a symbolic regression solution through the following key components.

Problem	Shape of X	Shape of y	Variables
1	(1, 500)	(500,)	['x0']
2	(3, 5000)	(5000,)	['x0', 'x1', 'x2']
3	(3, 5000)	(5000,)	['x0', 'x1', 'x2']
4	(2, 5000)	(5000,)	['x0', 'x1']
5	(2, 5000)	(5000,)	['x0', 'x1']
6	(2, 5000)	(5000,)	['x0', 'x1']
7	(2, 5000)	(5000,)	['x0', 'x1']
8	(6, 50000)	(50000,)	['x0', 'x1', 'x2', 'x3', 'x4', 'x5']

Table 1.1: Data

1.4.1 Problem Difficulty Evaluation

The code starts with the evaluation of the difficulty of each problem based on factors such as the number of variables, number of samples, and variability in the target values (y). This allows to understand the complexity of the task before proceeding with the symbolic regression and with the choice of problem's parameters.

```
def evaluate_problem_difficulty(problems):
    difficulty_scores = []
    for i, problem in enumerate(problems):
        X, y, variables = problem["X"], problem["y"], problem["variables
            "]
        num_vars = len(variables)
        num_samples = X.shape[1]
        variability = np.std(y)
        difficulty = (num_vars * 2) + (num_samples / 5000) + (variability
             / (np.median(y) + 1e-8))
        difficulty_scores.append((i + 1, difficulty, X, y, variables))
    return difficulty_scores
```

1.4.2 Tree Representation (Node Class)

Mathematical expressions are designed as trees and each node containing either a constant value, a variable, or a operator (e.g., addition, multiplication, trigonometric functions). A Node class is defined to save the data (constant/-variable/operator) and its children (sub-trees). In particular, there are several implemented methods:

- ***evaluate***: Calculates the value of the tree/expression by recursively evaluating its children. It handles errors (e.g., division by zero) by returning a penalty value

- ***to_formula***: Returns a string representation of the expression, formatted as a mathematical formula, for a better visualization

- ***extract_variables***: collects and returns all variables used in the tree

```
class Node:
    def __init__(self, data, children=None):
        self.data = data # Operator, variable, or constant
        self.children = children if children else []

    def evaluate(self, variables):
        """ Evaluates the mathematical expression represented by the tree
             """
        if isinstance(self.data, (int, float)):
            return self.data # Constant value
        elif isinstance(self.data, str):
            return variables[self.data] # Variable lookup
        elif callable(self.data):
            with np.errstate(all='ignore'):
                try:
                    if len(self.children) == 1:
                        return self.data(self.children[0].evaluate(
                            variables))
                    elif len(self.children) == 2:
                        return self.data(self.children[0].evaluate(
                            variables),
                           self.children[1].evaluate(variables))
                except (FloatingPointError, ValueError, ZeroDivisionError)
                    :
                    return float('inf') # Penalize invalid expressions

    def to_formula(self):
        if isinstance(self.data, (int, float)):
            return str(round(self.data, 2))
        if isinstance(self.data, str):
            return self.data # Variable name
        if callable(self.data):
            if len(self.children) == 1:
                return f"{UNARY_SYMBOLS.get(self.data, '?')}({self.
                    children[0].to_formula()})"
            elif len(self.children) == 2:
                return f"({self.children[0].to_formula()} {OP_SYMBOLS.get(
                    self.data, '?')} {self.children[1].to_formula()})"
        return "?"

    def extract_variables(self):
        if isinstance(self.data, str): # If the node is a variable
            return [self.data]
        elif isinstance(self.data, (int, float)):
            return [] # Constants do not contain variables
        else:
            variables = []
            for child in self.children:
                variables.extend(child.extract_variables())
            return variables
```

1.4.3 Math operators and Protected Functions

Both unary and binary operators are included and to prevent numerical errors, protected versions of certain operators are implemented (e.g., *protected_div* to avoid division by zero, *protected_log* to handle log(0), and *protected_sqrt* to ensure the square root of non-negative numbers).

```
#Prevent division by zero
def protected_div(x, y):
    return np.divide(x, y + 1e-6)
#Avoid log(0) or negative numbers
def protected_log(x):
    return np.log(np.abs(x) + 1e-6)
#Ensure sqrt of non-negative values
def protected_sqrt(x):
    return np.sqrt(np.abs(x))

BINARY_OPS = [np.add, np.subtract, np.multiply, protected_div, np.power]
UNARY_OPS = [np.sin, np.cos, np.tan, protected_log, np.exp,
    protected_sqrt, np.abs]

OP_SYMBOLS = {np.add: '+', np.subtract: '-', np.multiply: '*',
    protected_div: '/', np.power: '^'}
UNARY_SYMBOLS = {np.sin: 'sin', np.cos: 'cos', np.tan: 'tan',
    protected_log: 'log',
                np.exp: 'exp', protected_sqrt: 'sqrt', np.abs: 'abs'}
```

1.4.4 Fitness Function

The fitness of each tree is evaluated based on how well it approximates the target values. In particular, the **Mean Squared Error (MSE)** between the predicted values (y_pred) and the actual target values (y) is used as a measure to evaluate performance. A penalty is added if inside the tree there are fewer variables than provided by the specific problem, which encourages the algorithm to select all the variables that the problem has.

```
def calculate_fitness(individual, variables, X, y, penalty=4.5):
    try:
        y_pred = np.array([individual.evaluate(dict(zip(variables, x)))
            for x in X.T])
        if np.any(np.isnan(y_pred)) or np.any(np.isinf(y_pred)):
            return float('inf') # Penalize invalid expressions
        used_variables = set(individual.extract_variables())
        #Add complexity if not all variables are used
        complexity_penalty = penalty * (len(variables) - len(
            used_variables)) if len(used_variables) < len(variables) else
             0
        return np.mean((y - y_pred) 2) + complexity_penalty
    except (FloatingPointError, ValueError, TypeError):
        return float('inf')
```

1.4.5 Genetic Operators: Parent Selection, Crossover and Mutation

Parent Selection. Tournament Selection allows a selection of individuals for crossover and mutation by choosing in a random way some individuals from the population, depending on the tournament_size. The tree with the lowest fitness value in this selected set will be the selected individual and it will be returned as a parent for the next generation.

```
\begin{minted}[fontsize=\small, breaklines]{python}
def tournament_selection(population, fitness_dict, tournament_size):
    tournament = random.sample(population, tournament_size)
    winner = min(tournament, key=lambda ind: fitness_dict[ind])
    return winner
```

Crossover. The crossover operator performs a swap of sub-trees between two parent trees. It creates deep copies of both parents, selects random sub-trees from each, and swaps them by exchanging their data and children. The resulting trees are returned as offspring.

```
def crossover(parent1, parent2):
    child1, child2 = copy.deepcopy(parent1), copy.deepcopy(parent2)
    node1, node2 = random.choice(child1.children), random.choice(child2.
        children)
    node1.data, node2.data = node2.data, node1.data
    node1.children, node2.children = node2.children, node1.children
    return child1
```

Mutation. The mutate function changes its mutation strength, based on the current generation. In the early generations, there is an increase in the mutation strength to promote exploration, thus the mutation becomes more aggressive, by making more significant changes to the tree. Instead, when the current generation is high, the mutation strength decreases, encouraging more controlled adjustments to refine and optimize the solutions. The function recursively applies the mutation to all child nodes of the tree, spreading the mutation throughout the structure.

```
def mutate(node, variables, generation, max_depth=3, agressive_rate=0.7,
    top=30):
    #Mutates a node in the tree, adjusting mutation intensity over
        generations
    if generation >= top:
        agressive_rate = max(0.1, agressive_rate * 0.5) # Reduce mutation
            in later generations
    else:
        agressive_rate = min(1.0, agressive_rate * 1.2) # Increase
            mutation in early generations

    if random.random() < agressive_rate:
        if isinstance(node.data, str):
            node.data = random.choice(variables) # Change variable
        elif isinstance(node.data, (int, float)):
```

```
            node.data *= random.uniform(0.5, 1.5) # Perturb numeric values
        elif callable(node.data):
            node.data = random.choice(UNARY_OPS if len(node.children) == 1
                 else BINARY_OPS)

    if node.children and generation < top and random.random() <
        agressive_rate:
        if random.random() < 0.5:
            return copy.deepcopy(generate_random_tree(max_depth, variables
                )) # Replace subtree

    for i in range(len(node.children)):
        node.children[i] = mutate(node.children[i], variables, generation
            , max_depth, agressive_rate, top)

    return node
```

1.4.6 Tree and Population Generation

The initial population of individuals is generated in a random way, using the **grow**, **full** or **half-and-half** approaches to build the trees. These methods allow to control the depth and structure of the trees and in particular with *grow* generates trees with variable depth, while *full* producing trees of fixed depth.

```
def generate_population(pop_size, max_depth, variables, method='
    half_and_half'):
    return [generate_random_tree(max_depth, variables, method=method) for
         _ in range(pop_size)]

def generate_random_tree(max_depth, variables, depth=0, method='grow'):
    constant_prob = 0.4
    if depth >= max_depth or (depth > 0 and random.random() < 0.2):
        if random.random() < 0.7:
            return Node(random.choice(variables)) #variable node
        else:
            return Node(round(random.uniform(-10, 10), 2)) #random
                constant node between -10 and 10

    # If method is 'full', generate fully-grown trees
    if method == 'full':
        if random.random() < 0.5:
            op = random.choice(UNARY_OPS)
            return Node(op, [generate_random_tree(max_depth, variables,
                depth + 1, method='full')])
        else:
            op = random.choice(BINARY_OPS)
            if random.random() < constant_prob:
                #Randomly decide which child will be a constant
                if random.random() < 0.5:
```

```
                #Left child is a constant
                left_child = Node(random.uniform(-10, 10))
                right_child = generate_random_tree(max_depth, variables
                    , depth + 1, method='full')
            else:
                #Right child is a constant
                left_child = generate_random_tree(max_depth, variables,
                     depth + 1, method='full')
                right_child = Node(random.uniform(-10, 10))
        else:
            #No constants
            left_child = generate_random_tree(max_depth, variables,
                depth + 1, method='full')
            right_child = generate_random_tree(max_depth, variables,
                depth + 1, method='full')

        return Node(op, [left_child, right_child])

# If method is 'grow', generate trees with variable depth
elif method == 'grow':
    if random.random() < 0.5:
        op = random.choice(UNARY_OPS)
        return Node(op, [generate_random_tree(max_depth, variables,
            depth + 1, method='grow')])
    else:
        op = random.choice(BINARY_OPS)
        if random.random() < constant_prob:
            if random.random() < 0.5:
                #Left child is a constant
                left_child = Node(random.uniform(-10, 10))
                right_child = generate_random_tree(max_depth, variables
                    , depth + 1, method='grow')
            else:
                #Right child is a constant
                left_child = generate_random_tree(max_depth, variables,
                     depth + 1, method='grow')
                right_child = Node(random.uniform(-10, 10))
        else:
            #No constants
            left_child = generate_random_tree(max_depth, variables,
                depth + 1, method='grow')
            right_child = generate_random_tree(max_depth, variables,
                depth + 1, method='grow')
        return Node(op, [left_child, right_child])

# If method is 'half_and_half', mix 'full' and 'grow' methods
elif method == 'half_and_half':
    if random.random() < 0.6:
        return generate_random_tree(max_depth, variables, depth,
            method='full')
```

```
        else:
            return generate_random_tree(max_depth, variables, depth,
                method='grow')
```

1.4.7 Evaluation

The *evolve* function processes the genetic programming process in a specified number of generations (*epochs*) to evolve a population of candidate solutions. It employs a combination of elitism, crossover, and mutation operators to iteratively improve the population.

1. **Initialization**: The fitness of each individual in the initial population is calculated using the provided fitness function, which evaluates the individual's performance based on the selected problem.

2. **Elitism**: The top 10% of the population, based on the fitness value, is preserved into the next generation, ensuring that the best solutions are always retained.

3. **Reproduction**: The rest of the population is generated through a combination of crossover and mutation operations, ensuring a mix of exploration and exploitation.

- Crossover: Two parents are selected through tournament selection and they are combined to produce offspring. The crossover probability determines how frequently this operation occurs and the values of this parameter can change during the process.

- Mutation: Mutation is applied to single parents with a certain probability. It introduces random variations by modifying the structure of the tree.

4. **Fitness Calculation**: After the new individuals are generated, their fitness is evaluated, while the fitness values of the elite individuals remains unchanged.

5. **Adaptation**:

- **Crossover and Mutation Adjustment**: If no improvement in the best fitness is observed after a set number of generations, the probability for crossover decreases and so consequently the mutation rate increases, to introduce more diversity and avoid stagnation.

- **Population Regeneration**: If no improvement in the best fitness is observed over a specified period, a new population is generated from scratch while maintaining the best individual. This mechanism helps maintain diversity and encourages exploration.

Throughout the evolution process, the algorithm balances exploration and exploitation through dynamic adjustments in mutation and crossover rates and regenerating the population. This helps ensure a robust search for optimal solutions while maintaining diversity and avoiding premature convergence.

At the end of the process, the best individual and its corresponding fitness value, along with a history of fitness evolution, are returned.

```
def evolve(population, epochs, X, y, variables, depth=3, cross_prob=0.7,
    tournament_size = 8, penalty=4.5):
```

```
fitness_history = []
#Counter for epochs with no improvement in fitness
no_improvement_count = 0
#Counter for epochs with no significant change in the best fitness
no_change = 0

#Calculate the initial fitness for all individuals and store it
fitness_dict = {ind: calculate_fitness(ind, variables, X, y, penalty)
     for ind in population}

best_individual = min(fitness_dict, key=fitness_dict.get)
best_fitness = fitness_dict[best_individual]

fitness_history.append(best_fitness)

#Evolving the population over multiple generations
for gen in tqdm(range(epochs), desc="Evolving Generations"):

    #Elitism: Select the best individuals to carry over to the new
        generation
    sorted_population = sorted(fitness_dict.keys(), key=lambda ind:
        fitness_dict[ind])
    elite_size = max(1, len(population) // 10) #Keep top 10% as
        elites
    new_population = sorted_population[:elite_size] #Keep the best
        individuals

    while len(new_population) < len(population):
        if random.random() < cross_prob:
            #Crossover: Select two parents using tournament selection
            p1 = tournament_selection(sorted_population[:len(
                population)//2], fitness_dict, tournament_size)
            p2 = tournament_selection(sorted_population[:len(
                population)//2], fitness_dict, tournament_size)
            offspring = crossover(p1, p2)
        else:
            #Mutation: Select one parent using tournament selection
            p = tournament_selection(sorted_population[:len(population
                )//2], fitness_dict, tournament_size)
            offspring = mutate(p, variables, depth)

        new_population.append(offspring)

    #Calculate fitness for the new individuals (excluding elite
        individuals)
    fitness_dict = {ind: (fitness_dict[ind] if ind in
        sorted_population[:elite_size] else calculate_fitness(ind,
        variables, X, y, penalty)) for ind in new_population}

    #Identify the current best individual and its fitness
```

```
        current_best = min(fitness_dict, key=fitness_dict.get)
        current_best_fitness = fitness_dict[current_best]

        #Update the best individual if there is an improvement in fitness
        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_individual = copy.deepcopy(current_best)
            no_improvement_count = 0
            no_change = 0
        else:
            no_improvement_count += 1
            no_change += 1

        fitness_history.append(best_fitness)


        formula = best_individual.to_formula() if best_individual else "N
            /A"
        print(f"Generation {gen}: Best fitness = {best_fitness}, Formula:
             {formula}")

        #Adapt crossover and mutation probabilities if no improvement for
             10 generations
        if no_improvement_count >= 10:
            no_improvement_count = 0
            cross_prob = max(0.4, cross_prob - 0.05)
            print(f"Adapting parameters: Less Crossover = {cross_prob},
                more mutation")

        #Regenerate the population for diversity if there are no
            significant changes for 6 generations
        if no_change >= 6:
            print("Regenerating population for diversity!")
            new_population = generate_population(len(population), depth,
                variables)
            new_population.append(best_individual) #Mantain the best
                individual
            cross_prob = min(0.7, cross_prob + 0.05) #bring back crossover
                 probability
            no_change = 0

        # Update the population for the next generation
        population = new_population

    return best_individual, best_fitness, fitness_history
```

1.4.8 Parameter Tuning

In the evolution process, several parameters are included and need to be chosen effectively for each specific problem. The **epochs** parameter says how many iterations will be performed during the entire evolution and the **population size** defines how many trees are in each generation. **Crossover rates** control how often crossover occurs differently from mutation and **tournament size** is used in tournament selection to determine how many individuals will participate in each tournament when selecting parents for crossover or mutation operators. **Depth** imposes the maximum depth of generated trees and finally **penalty** reflects the added penalty on the fitness score, if trees do not contain all variables of the problem.

1.4.9 Visualization and Result Analysis

The progress of the optimization can be observed by monitoring the best fitness score (MSE) over generations. A plot of the fitness history is displayed, showing how the error decreases as the population evolves toward a better solution. The best formula discovered by the genetic algorithm is printed, along with its fitness score. In the following, we can observe an example of testing the process for problem 1:

```
problem_id = 1
data = np.load(f"data/problem_{problem_id}.npz")
X = data['x']
y = data['y']

print("Shape di X:", X.shape)
print("Shape di y:", y.shape)

num_vars = X.shape[0]
variables = [f"x{i}" for i in range(num_vars)]
print("Variales:", variables)
print()

initial_population = generate_population(pop_sizes[problem_id-1],
    max_depths[problem_id-1], variables)
best_formula, best_fitness, fitness_history = evolve(
                                                initial_population,
                                                    epochs_list[
                                                    problem_id-1], X, y,
                                                    variables,
                                                max_depths[problem_id-1],
                                                    cross_prob,
                                                tournament_sizes[
                                                    problem_id-1],
                                                    penalties[problem_id
                                                    -1])
```

```
print()
print(f"Best Formula found: {best_formula.to_formula()}")
print(f"Fitness: {best_fitness}")
plt.plot(fitness_history)
plt.xlabel("Generation")
plt.ylabel("Fitness (MSE)")
plt.title("Fitness history")
plt.show()
```

1.4.10 Results

Simple problems with a single formula (Problem 1): For simpler problems with a single formula, the algorithm achieved excellent results with fitness values close to zero and rapid convergence. The performance was very efficient, reaching optimal solutions quickly.

Problems with no more than 2 variables: For moderately complex problems with up to 2 variables, execution time is acceptable. Although the results were not always optimal, the algorithm showed decent evolution and good progress, with some decline in fitness observed over time.

Problems with 3 or more variables (Problem 2, 3 and 8): For more complex problems with 3 or more variables, the algorithm required much longer execution times. The results were not fully optimal, but a good descent in fitness was achieved. These problems consumed a considerable amount of computational resources.

In the following images, we can observed some of this results obtain, for each of the problem.

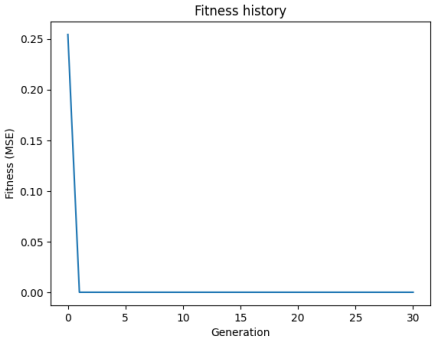

Figure 1: Problem 1

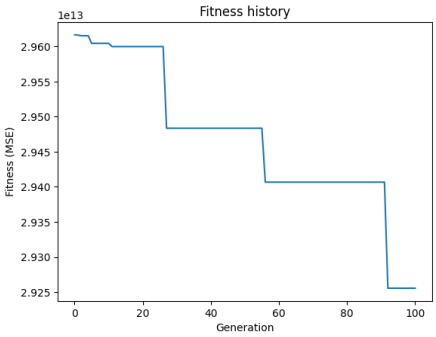

Figure 2: Problem 2

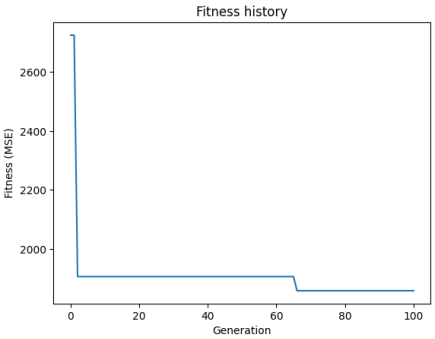

Figure 3: Problem 3

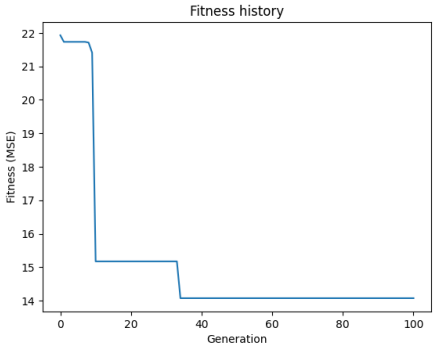

Figure 4: Problem 4

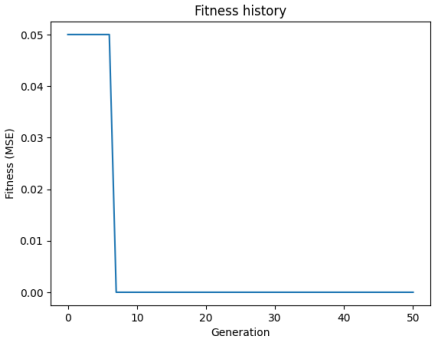

Figure 5: Problem 5

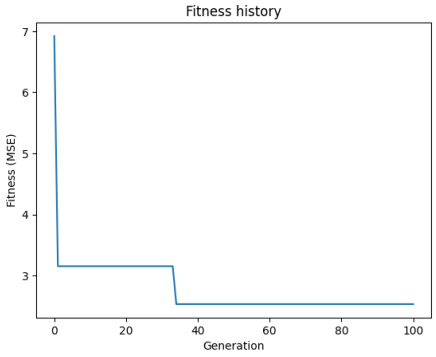

Figure 6: Problem 6

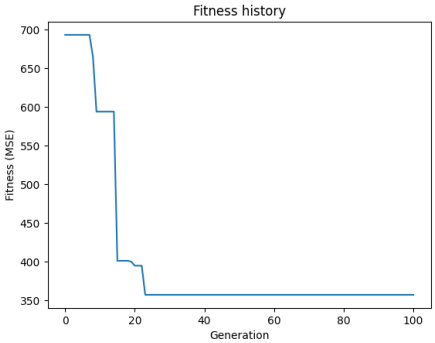

Figure 7: Problem 7

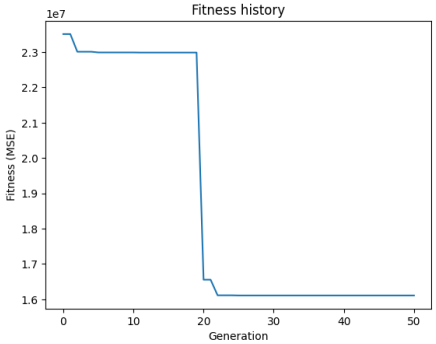

Figure 8: Problem 8