Exploartion_17 : Tree-Based Genetic Programming for Polynomial Regression

this is for `matlab_translation_symbolic_regression_18.m` file implementation

Mohammed Tawheed Khater

2024-06-23

This document is intended to be a simpler alternative for translating symbolic regression code from Python to MATLAB. It focuses on a more straightforward approach, as translating the Node and Tree classes from the Python code on GitHub (by dyckia, titled "Genetic-Programming-Polynomial-Regression") proved challenging.

a complete example working matlab file `matlab_translation_symbolic_regression_18.m` will be sent with this document via whatsapp.

later i will upload it to github with proper permenant link.

here is link the translated file : - matlab_translation_symbolic_regression_18.m

here is link to the training data used : - A4_trainingSamples.txt

Table of Contents

download this doc as pdf or word

Other Formats

PDF

MS Word

Introduction

Symbolic regression, a technique used to identify mathematical expressions that best fit a given dataset, is a complex task often tackled using genetic programming (GP). Genetic programming mimics natural evolution to optimize candidate solutions, represented as mathematical expressions. This review explores the implementation of a genetic algorithm for symbolic regression using MATLAB, delving into the detailed structure of the genetic algorithm and the representation of mathematical expressions as tree structures. The following sections will elaborate on the genetic algorithm's key components, the tree structure for representing mathematical expressions, and the nuances of the MATLAB implementation.

Genetic Algorithm Implementation

The genetic algorithm (GA) for symbolic regression involves several critical steps: population initialization, selection, crossover, mutation, and fitness evaluation. Each of these steps is meticulously implemented in MATLAB to evolve mathematical expressions that best fit the provided data.

1. Population Initialization

Population initialization is achieved using the ramped half-and-half method, which ensures a diverse initial population. This method generates trees with varying depths and structures, promoting genetic diversity.

```matlab
function pop = initialPopulation(pop_size, max_depth, terminal_set, function_set)
    pop = cell(1, pop_size);
    for i = 1:pop_size
        method = 'full';
        if i > (pop_size / 2)
            method = 'grow';
        end
        pop{i} = growTree(max_depth, method, 0, terminal_set, function_set);
    end
    pop = pop(randperm(length(pop)));  % Randomize the population
end
```

The `growTree` function generates individual trees using either the “full” method, creating fully populated trees to the maximum depth, or the “grow” method, generating trees with random depths.

2. Selection

Tournament selection is employed to choose individuals for reproduction. This method ensures that individuals with better fitness are more likely to be selected, driving the population towards better solutions.

```matlab
function selected = tournamentSelection(pop, tour_size, x, y, rf, terminal_set)
    selected_ind = randperm(length(pop), tour_size);
    selected = cell(tour_size, 2);
    for i = 1:tour_size
        tree = pop{selected_ind(i)};
        fitness = computeError(tree, x, y, rf, terminal_set);
        selected{i, 1} = tree;
        selected{i, 2} = fitness;
    end
    selected = sortrows(selected, 2);  % Sort based on fitness (ascending)
end
```

Tournament selection enhances the likelihood of selecting fitter individuals while maintaining some level of genetic diversity, which is crucial for avoiding local optima.

3. Crossover

Crossover involves exchanging subtrees between pairs of parent trees to produce offspring with mixed characteristics. This operation introduces variability and combines the strengths of both parents.

```
function [tree1, tree2] = crossover(tree1, tree2, c_rate)
    if rand() <= c_rate
        swap_point1 = randi([0, countNodes(tree1) - 1]);
        swap_point2 = randi([0, countNodes(tree2) - 1]);
        subtree1 = getSubtree(tree1, swap_point1);
        subtree2 = getSubtree(tree2, swap_point2);
        tree1 = setSubtree(tree1, swap_point1, subtree2);
        tree2 = setSubtree(tree2, swap_point2, subtree1);
    end
end
```

4. Mutation

Mutation introduces random changes in the tree structure, which is essential for maintaining genetic diversity and exploring new solutions.

```
function tree = mutate(tree, m_rate, max_depth, terminal_set, function_set)
    if rand() <= m_rate
        mut_point = randi([0, countNodes(tree) - 1]);
        subtree = growTree(max_depth, 'grow', 0, terminal_set, function_set);
        tree = setSubtree(tree, mut_point, subtree);
    end
end
```

5. Fitness Evaluation

The fitness of each tree is evaluated based on its ability to predict target values accurately. The fitness metric used is the root mean square error (RMSE), which quantifies the difference between the predicted and actual values.

```
function error = computeError(tree, x, y, rf, terminal_set)
    error_sum = 0;
    n = length(x);
    for i = 1:n
        error_sum = error_sum + (computeTree(tree, x(i), y(i), terminal_set) - rf(i))^2;
    end
    error = sqrt(error_sum / n);
end
```

Tree Structure for Mathematical Expressions

In genetic programming, mathematical expressions are represented as tree structures, where nodes represent functions or terminals. Terminals can be variables (e.g., `x`, `y`) or constants (e.g., `1`, `2`), while functions are operators (e.g., `+`, `-`, `*`, `/`).

Tree Representation

Each tree node is a structure with a value (operator or terminal) and pointers to its left and right child nodes.

```matlab
function node = createNode(value, left, right)
    node = struct('value', value, 'left', left, 'right', right);
end
```

Tree Generation

Trees are generated using the `growTree` function, which recursively builds trees up to a specified maximum depth. The tree generation can follow the "full" method, creating fully populated trees, or the "grow" method, producing trees with variable depths.

```matlab
function tree = growTree(max_depth, method, depth, terminal_set, function_set)
    if strcmp(method, 'full') && depth < max_depth
        left = growTree(max_depth, method, depth + 1, terminal_set, function_set);
        right = growTree(max_depth, method, depth + 1, terminal_set, function_set);
        value = drawValue('f', terminal_set, function_set);
    elseif strcmp(method, 'grow') && depth < max_depth
        value = drawValue('tf', terminal_set, function_set);
        if any(strcmp(value, terminal_set))
            tree = createNode(value, [], []);
            return;
        end
        left = growTree(max_depth, method, depth + 1, terminal_set, function_set);
        right = growTree(max_depth, method, depth + 1, terminal_set, function_set);
    else
        value = drawValue('t', terminal_set, function_set);
        left = [];
        right = [];
    end
    tree = createNode(value, left, right);
end
```

Tree Evaluation

The `computeTree` function recursively evaluates the tree by computing the value of each subtree and applying the respective operator.

```
function result = computeTree(tree, x, y, terminal_set)
    if any(strcmp(tree.value, terminal_set))
        result = eval(tree.value);
    else
        left_val = computeTree(tree.left, x, y, terminal_set);
        right_val = computeTree(tree.right, x, y, terminal_set);
        if strcmp(tree.value, '/') && right_val == 0
            result = 1; % Avoid division by zero
        else
            result = eval([num2str(left_val) tree.value num2str(right_val)]);
        end
    end
end
```

MATLAB Implementation Details

The MATLAB code employs several programming techniques to ensure the efficient implementation of the genetic algorithm:

Modular Functions

The code is organized into modular functions, each responsible for a specific task, enhancing readability and maintainability.

Randomization

Functions such as `randperm` and `randi` introduce randomness in the selection, crossover, and mutation processes, which is vital for maintaining genetic diversity.

Structured Logging

Execution progress and timing are logged using the `logMessage` function, providing valuable insights into the algorithm's performance.

```
function logMessage(message)
    timestamp = datestr(now, 'yyyy-mm-dd HH:MM:SS');
    fprintf('[%s] %s\n', timestamp, message);
end
```

Visualization

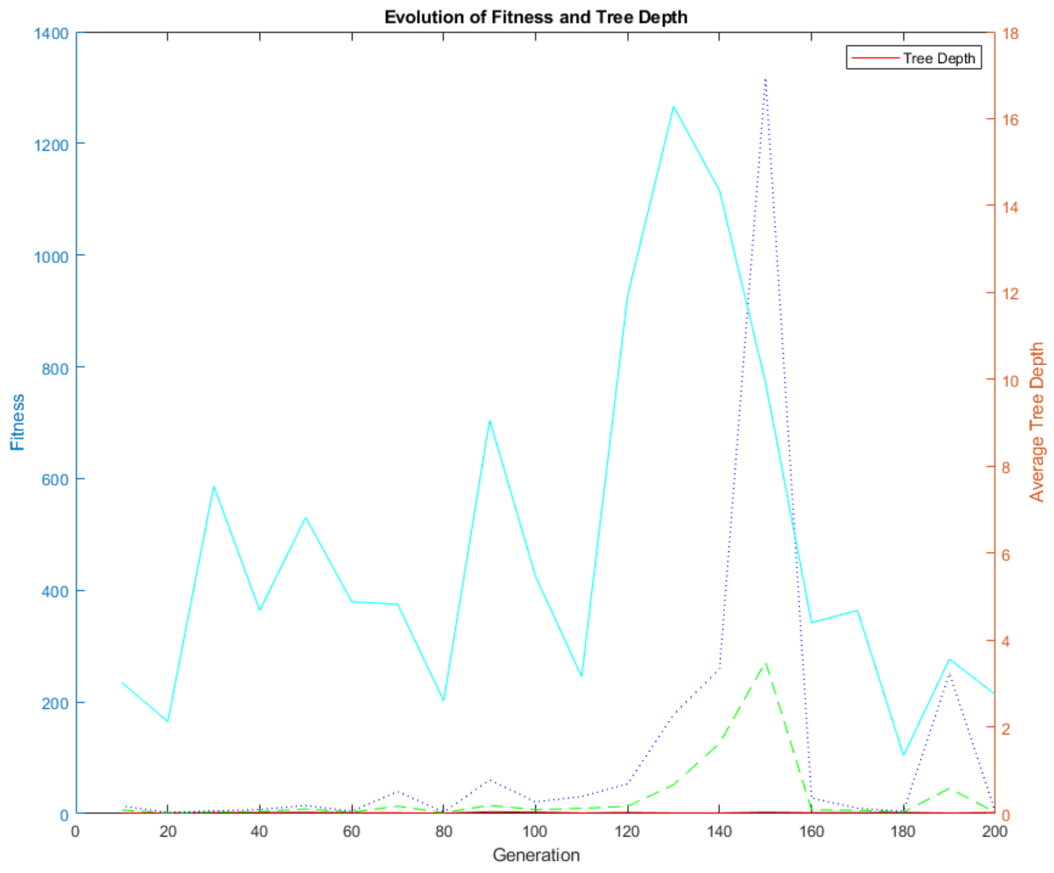

Figure 1: exanple plot with 200 generations

The plot visualizes the evolutionary process of symbolic regression using a genetic algorithm , highlighting the improvement in fitness (error reduction) and changes in solution complexity (tree depth) across generations. This approach aims to find mathematical expressions (symbolic models) that best fit the training data (A4_trainingSamples.txt). Each line provides critical insights into the algorithm's performance and the nature of solutions generated over time.

The `plotFigure` function visualizes the evolution of fitness and tree depth over generations, aiding in the interpretation of the algorithm's progress.

```
function plotFigure(best_fit, avg_fit, worst_fit, tree_depth, gen)
    generations = 10:10:gen;
    figure;
    yyaxis left
    plot(generations, best_fit, 'r', generations, avg_fit, 'g', generations, worst_fit, 'b')
    xlabel('Generation');
    ylabel('Fitness');
    legend('Best', 'Average', 'Worst');

    yyaxis right
    plot(generations, tree_depth, 'c');
    ylabel('Average Tree Depth');
    legend('Tree Depth');

    title('Evolution of Fitness and Tree Depth');
end
```

Plot Explained:

Fitness Evolution (Left Y-axis):

Red Line (Best Fitness): Represents the fitness (error) of the best individual in each generation. Green Line (Average Fitness): Shows the average fitness (error) of the population in each generation. Blue Line (Worst Fitness): Indicates the fitness (error) of the worst individual in each generation. These lines show how the fitness (error) values change over generations. The goal of the algorithm is typically to minimize this fitness value, as it represents the error in predicting the output (rf) based on the input (x, y).

Tree Depth Evolution (Right Y-axis):

Cyan Line (Average Tree Depth): Displays the average depth of the trees in the population for each generation. The depth of the trees is an indicator of their complexity. Genetic programming often evolves trees of varying depths, and monitoring the average tree depth helps in understanding the complexity of the solutions found over generations.

X-axis (Generation):

Each point on the x-axis represents a specific generation number (10:10:gen), where gen is the total number of generations specified in the code. Interpretation: Fitness Lines (Red, Green, Blue): Ideally, you want to see the red (best fitness) line decreasing over generations, indicating improvement in the best solution found. The green (average fitness) and blue (worst fitness) lines provide context on the overall performance of the population.

Tree Depth Line (Cyan): Monitoring the tree depth helps in understanding if the algorithm is converging towards simpler or more complex solutions. An increase in average tree depth might indicate overfitting or increasing complexity of solutions.

Specific Colors in the Plot: Green Line: Represents the average fitness (error) of the population. Blue Line: Represents the worst fitness (error) in the population. Cyan Line: Represents the average depth of trees in the population. These lines collectively show how the genetic algorithm progresses in terms of fitness (error minimization) and the complexity of the evolved solutions (average tree depth).

The plot helps in understanding how the fitness of the population improves over time and how the complexity of the solutions (tree depth) changes.

Conclusion

The MATLAB implementation of genetic programming

for symbolic regression is a robust approach that leverages the power of evolutionary algorithms to discover optimal mathematical expressions. Key features include structured functions, effective randomization, detailed logging, and clear visualization. This review highlights the importance of genetic diversity, modular programming, and the use of tree structures for representing mathematical expressions. These elements collectively enhance the effectiveness of symbolic regression using genetic algorithms.

References

this part is still uderconstruction to be continued later