Project Report: Studies of Perceptron Learning and Adaline

Implementations: https://github.com/dcazih/regression-trees

Dmitri Azih and Dylan Rivas

1. Implementing a Binary Regression Tree Using CART Algorithm

In this task, we built a regression tree class that allows the user to specify a maximum height or a leaf size that terminates the building of the tree. This class finds the best feature to split a dataset based on a proper split value. The regression tree takes in a dataset, a maximum height, a leaf size, and a control variable that tells whether the height or leaf size is limited. There is also a Node class that contains a feature, threshold, value, its left node, and its right node, as well as the depth. The regression tree class has its constructor function, a function to calculate the sum squared errors, a function to find the best split, a function to build the regression tree, a function to traverse the tree, a function to predict the value for the input, and lastly a decision path function that displays the decision path of a given input value.

The algorithm calculates the sum squared error of a sample from the mean of the samples in order to decide the best split. In order to make this decision of the best split, the algorithm loops over each feature, and then tries each split point. We know if a split is good when it has a low sum squared error. The best split is tracked until the loop is finished. That split has the best dimension, or feature, as well as the best split value for that feature.

The following is the pseudocode of tree construction:

```
function BuildTree(X, y, depth)
      if (all X same):
            return leaf node value with mean of y
      if the height limit is reached:
            return leaf node value with mean of y
      if the leaf size limit is reached:
            return leaf node value with mean of y
      (feature, threshold) ← Best split of (X, y)
      if no error reduction:
            return leaf node value with mean of y
      Split X and y
            left is X feature <= threshold
            right is X feature > threshold
      if left or right is empty
            return leaf node value with mean of y
      BuildTree at left child
      BuildTree at right child
      Return node with (feature, threshold, left, right)
```

2. Testing the Regression Tree on Continuous Function Approximation

For this task, we tested the regression tree implementation by using the continuous function $y = 0.8\,sin(x - 1)$ over the domain x $\in$ [-3,3]. One test uses a tree that has no limitation to the tree. The next test uses a tree that has a limit on the height, which is half of the height of the tree with no limit. The last test uses three different trees that have limit to the leaf size, one with a size of 2, one with a size of 4, and the last a size of 8. The following is the test results of: the height of the tree, the test error, and the time cost of building the tree:

	Limitless Tree	Half Height Limit Tree	Leaf Size Limit: 2	Leaf Size Limit: 4	Leaf Size Limit: 8
Tree Height	15	7	14	12	9
Test Error	0.00119	0.00182	0.00189	0.00293	0.00933
Time Cost (s)	0.01093	0.00880	0.00922	0.00788	0.00662

The test results show us the comparisons between the trees. The test error is the best for the limitless tree, which shows that more height means better results, although it also has the largest time cost. On the other side of it, limiting a leaf size to 8 allows the time cost to be lowest, but the test error is the highest. The half height tree's results are very similar to limiting the leaf size to 2, although the leaf size limit has double the tree height. The leaf size limit at 4 is also more in the middle, as it has low test error and low time cost. The best option is all dependant on the goal of the tester.

3. Designing a Regression Tree Model for Multidimensional Dynamical Systems

For this task, we designed a method using our regression tree to approximate multidimensional functions. When the exact system model is not known, but just has a set of paired data of the form $((x_{k,}, v_k), (x_{k+1}, v_{k+1}))$, a regression tree-based model can be used to predict the next state of the system. We designed a method that looks at the system's transition function from the data, which are defined by the variables: position x, and velocity v. They are defined by the equations $x_{k+1} = x_k + 0.1v_k$ and $v_{k+1} = 10$, as

described in the task description. We generated 500 samples, with x being uniformly between 0 and 100, and v at 10. Then, we used one regression tree to learn the mapping from $[x_k, v_k] \rightarrow x_{k+1}$. Then, another regression tree learns the mapping to v_{k+1}. We also used a helper function called predict_next_state to use the results of the trees to predict the next state. In the current code, we used the current state of (99, 10) to predict the next state. It is predicted as [100.00737, 10], which just has an error of .00737. This shows that the prediction was well done based on the data we gave it.

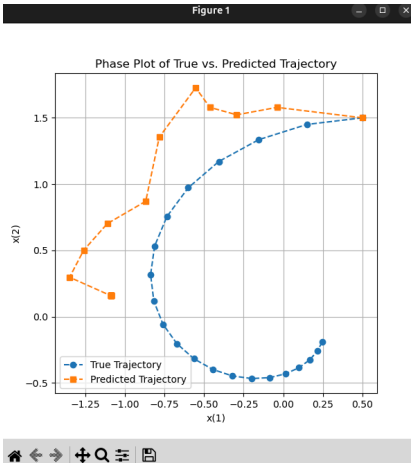
4. Evaluating Regression Tree Models on Case Studies

4.1 Case Study: Dynamic System

In this evaluation we created a function to generate the next set of samples of a form matching our system model ($x_{k+1}^{(1)} = 0.9x_k^{(1)} - 0.2x_k^{(2)},\ x_{k+1}^{(2)} = 0.2x_k^{(1)} + 0.9x_k^{(2)}$) in a specified range. Then using initial states ($x_0^{(1)} = 0.5,\ x_0^{(2)} = 1.5$) we predicted the future states when t = 1, 2, … 20 and plotted the predicted trajectories vs the true trajectories:

Results:

- At the end of evaluation, the final state **prediction** error was [-1.334, 0.350] while the **true** was [0.246, -0.191] an obvious dissonance which we can also see in the graph below:



- After tuning hyperparameters we found the best model parameters to be (**max depth=15, leaf_size=1**) to achieve an average MSE of 0.0442 as we can see from the results below (also showing test errors and time costs):

  === Dynamic System Hyperparameter Tuning ===
  Depth: 3 , Leaf: 1 | MSE: 0.4238 (x: 0.3880, z: 0.4596) | Time: 0.08s
  Depth: 3 , Leaf: 2 | MSE: 0.4238 (x: 0.3880, z: 0.4596) | Time: 0.08s

Depth: 3 , Leaf: 4 | MSE: 0.4238 (x: 0.3880, z: 0.4596) | Time: 0.08s
Depth: 3 , Leaf: 8 | MSE: 0.4238 (x: 0.3880, z: 0.4596) | Time: 0.10s
Depth: 3 , Leaf: 16 | MSE: 0.4238 (x: 0.3880, z: 0.4596) | Time: 0.08s
Depth: 5 , Leaf: 1 | MSE: 0.1269 (x: 0.1413, z: 0.1124) | Time: 0.12s
Depth: 5 , Leaf: 2 | MSE: 0.1269 (x: 0.1413, z: 0.1124) | Time: 0.12s
Depth: 5 , Leaf: 4 | MSE: 0.1269 (x: 0.1413, z: 0.1124) | Time: 0.14s
Depth: 5 , Leaf: 8 | MSE: 0.1269 (x: 0.1413, z: 0.1124) | Time: 0.13s
Depth: 5 , Leaf: 16 | MSE: 0.1269 (x: 0.1413, z: 0.1124) | Time: 0.13s
Depth: 7 , Leaf: 1 | MSE: 0.0606 (x: 0.0569, z: 0.0643) | Time: 0.18s
Depth: 7 , Leaf: 2 | MSE: 0.0606 (x: 0.0569, z: 0.0643) | Time: 0.18s
Depth: 7 , Leaf: 4 | MSE: 0.0606 (x: 0.0569, z: 0.0643) | Time: 0.18s
Depth: 7 , Leaf: 8 | MSE: 0.0606 (x: 0.0569, z: 0.0643) | Time: 0.17s
Depth: 7 , Leaf: 16 | MSE: 0.0606 (x: 0.0569, z: 0.0643) | Time: 0.18s
Depth: 10 , Leaf: 1 | MSE: 0.0446 (x: 0.0416, z: 0.0475) | Time: 0.25s
Depth: 10 , Leaf: 2 | MSE: 0.0446 (x: 0.0416, z: 0.0475) | Time: 0.22s
Depth: 10 , Leaf: 4 | MSE: 0.0446 (x: 0.0416, z: 0.0475) | Time: 0.24s
Depth: 10 , Leaf: 8 | MSE: 0.0446 (x: 0.0416, z: 0.0475) | Time: 0.24s
Depth: 10 , Leaf: 16 | MSE: 0.0446 (x: 0.0416, z: 0.0475) | Time: 0.25s
Depth: 15 , Leaf: 1 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.26s
Depth: 15 , Leaf: 2 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.23s
Depth: 15 , Leaf: 4 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.26s
Depth: 15 , Leaf: 8 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.27s
Depth: 15 , Leaf: 16 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.23s
Depth: 20 , Leaf: 1 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.24s
Depth: 20 , Leaf: 2 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.24s
Depth: 20 , Leaf: 4 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.28s
Depth: 20 , Leaf: 8 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.26s
Depth: 20 , Leaf: 16 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.23s
Depth: None, Leaf: 1 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.23s
Depth: None, Leaf: 2 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.24s
Depth: None, Leaf: 4 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.25s
Depth: None, Leaf: 8 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.24s
Depth: None, Leaf: 16 | MSE: 0.0442 (x: 0.0410, z: 0.0473) | Time: 0.23s

Best Parameters:
Depth: 15, Leaf Size: 1
Average MSE: 0.0442
Build Time: 0.26s

Analysis:

From the results we can observe that the model seems to capture the general rotational behavior of the system model but not exactly most likely due to compounding inaccuracies that build up each future prediction. This experiment

shows a regression tree's inability to maintain precise state information, its sensitivity and overall limitations to perform proper dynamic system modeling.

4.2 Case Study 2: Program System

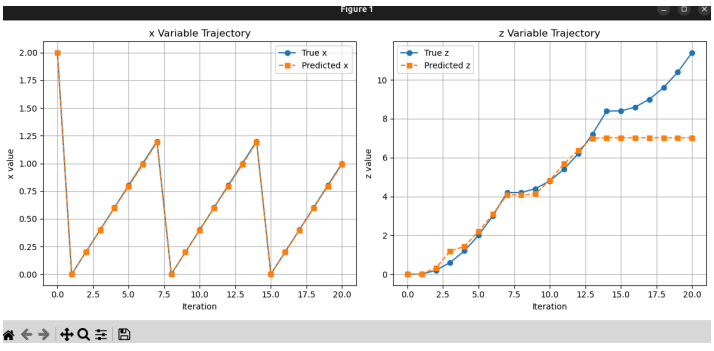
In this evaluation we created a regression tree-based model to predict the next state for the program:

```
def func(x):
    z = 0
    for _ in range(20):
        if x > 1:
            x = 0
        else:
            x = x + 0.2
        z = z + x
```

Given initial states x_0 = 2 and z_0= 0 we predicted the future states for t=1, 2, … 20 and compared the predictive trajectories with the actual program's trajectories.

Results:

- This graph shows the true trajectory for both the x and z variable against the regression tree's prediction:



- After tuning hyperparameters we found the best model parameters to be (**max depth=10, leaf_size=1**) to achieve an average MSE of 0.045; as seen in the results below (also showing test errors and time costs):

  === Program System Hyperparameter Tuning ===

Depth: 3 , Leaf: 1 | MSE: 0.5662 (x: 0.0416, z: 1.0909) | Time: 0.16s
Depth: 3 , Leaf: 2 | MSE: 0.5662 (x: 0.0416, z: 1.0909) | Time: 0.14s
Depth: 3 , Leaf: 4 | MSE: 0.5662 (x: 0.0416, z: 1.0909) | Time: 0.14s
Depth: 3 , Leaf: 8 | MSE: 0.5662 (x: 0.0416, z: 1.0909) | Time: 0.14s
Depth: 3 , Leaf: 16 | MSE: 0.5662 (x: 0.0416, z: 1.0909) | Time: 0.14s
Depth: 5 , Leaf: 1 | MSE: 0.1637 (x: 0.0024, z: 0.3250) | Time: 0.23s
Depth: 5 , Leaf: 2 | MSE: 0.1637 (x: 0.0024, z: 0.3250) | Time: 0.24s
Depth: 5 , Leaf: 4 | MSE: 0.1637 (x: 0.0024, z: 0.3250) | Time: 0.24s
Depth: 5 , Leaf: 8 | MSE: 0.1637 (x: 0.0024, z: 0.3250) | Time: 0.27s
Depth: 5 , Leaf: 16 | MSE: 0.1637 (x: 0.0024, z: 0.3250) | Time: 0.27s
Depth: 7 , Leaf: 1 | MSE: 0.0729 (x: 0.0002, z: 0.1456) | Time: 0.34s
Depth: 7 , Leaf: 2 | MSE: 0.0729 (x: 0.0002, z: 0.1456) | Time: 0.34s
Depth: 7 , Leaf: 4 | MSE: 0.0729 (x: 0.0002, z: 0.1456) | Time: 0.33s
Depth: 7 , Leaf: 8 | MSE: 0.0729 (x: 0.0002, z: 0.1456) | Time: 0.33s
Depth: 7 , Leaf: 16 | MSE: 0.0729 (x: 0.0002, z: 0.1456) | Time: 0.32s
Depth: 10 , Leaf: 1 | MSE: 0.0454 (x: 0.0000, z: 0.0907) | Time: 0.45s
Depth: 10 , Leaf: 2 | MSE: 0.0454 (x: 0.0000, z: 0.0907) | Time: 0.42s
Depth: 10 , Leaf: 4 | MSE: 0.0454 (x: 0.0000, z: 0.0907) | Time: 0.41s
Depth: 10 , Leaf: 8 | MSE: 0.0454 (x: 0.0000, z: 0.0907) | Time: 0.42s
Depth: 10 , Leaf: 16 | MSE: 0.0454 (x: 0.0000, z: 0.0907) | Time: 0.44s
Depth: 15 , Leaf: 1 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.49s
Depth: 15 , Leaf: 2 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.53s
Depth: 15 , Leaf: 4 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.49s
Depth: 15 , Leaf: 8 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.49s
Depth: 15 , Leaf: 16 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.51s
Depth: 20 , Leaf: 1 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.46s
Depth: 20 , Leaf: 2 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.45s
Depth: 20 , Leaf: 4 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.44s
Depth: 20 , Leaf: 8 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.43s
Depth: 20 , Leaf: 16 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.43s
Depth: None, Leaf: 1 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.43s
Depth: None, Leaf: 2 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.43s
Depth: None, Leaf: 4 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.43s
Depth: None, Leaf: 8 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.43s
Depth: None, Leaf: 16 | MSE: 0.0457 (x: 0.0000, z: 0.0913) | Time: 0.43s

Best Parameters:
Depth: 10, Leaf Size: 1
Average MSE: 0.0454
Build Time: 0.45s

Analysis:

In both our graphical and mathematical results we see the model made great predictions for the x variable (MSE approx. = 0.0) but with a poorer z prediction

(MSE approx. = 0.09). This makes sense as the x variable has the simple rule of resetting to 0 when greater than 1 else add 0.2, which trees are able to learn perfectly. But since z depends on the cumulative sum of x, small x errors ended up accumulating in z, explaining its errors.

4.3 Hyperparameter Tuning

Below is a table revealing the best hyperparameters for each case study along with their test errors and time costs (it must be noted that a range of best hyperparameters were found with the same MSE for both cases and below is the first set of parameters found for each case with the lowest MSE):

System	Best Depth	Best Leaf	Avg MSE	x-MSE	z-MSE	Build Time
Case Study 1: Dynamic	15	1	0.0442	0.041	0.047	0.26s
Case Study 2: Program	10	1	0.0454	0.000	0.091	0.45s

5. Contributions

The following individuals contributed to solving the assigned problems:

- **Dylan Rivas**: Responsible for **Task 1, Task 2** and **Task 3**.
- **Dmitri Azih**: Responsible for **Task 4**.