Nolan Brady

Decision Tree Learning: CART Algorithm

Brief Overview:

Decision Tree Learning refers to predictive modeling algorithms that form trees to make predictions. For this project I focused on the CART or Classification and Regression Tree algorithm, which has been hugely popular for decades. The cart algorithm forms a tree from test data, at the aim of predicting classification values (can only be two different values,) for a given row of data.

History:

The concept of Decision Trees was first documented in the field of psychology in 1966, meant for exploring the human mind and the human concept of learning (Institute of Computing Science in the Poznań University of Technology). Later in 1972 the first classification tree appeared in the THIAD project, splitting data recursively to fit modals. In 1988 the first CART algorithm was invented, quickly becoming the standard for machine learning algorithms, continuing to be approved upon, “Main upgrades include truncating unnecessary trees, tunneling, and choosing the best tree version”(Explorium), and is still being improved upon today.

Intuition/Pseudocode: More specific the further down the intuition/Pseudocode is

Decision Tree:

Given a training dataset, a testing dataset, a max depth, and a minimum size; first build a decision tree from the training dataset with depth less than the minimum size, and leafs being

formed when a split dataset reaches a certain length. With that tree, plug in test data one row at a time, drilling down until reaching a leaf node representing the predicted classifier for that piece of data, adding it to a list which will be the resulting output of this algorithm.

```
Decision_Tree(train, test, max depth, min size)
  tree = build_tree(train, max depth, min size)
  predictions[]
  for each row in test
    predictions + predict(tree, row)
  return predictions
```

After feeding the training data to the build tree algorithm, the tree variable points to a tree capable of making predictions about a classification of a given row of data. The for loop simply iterates through the test data, drilling down the build tree until it reaches a leaf node, giving us a prediction. All of the predictions are added to a list and then returned.

Build Tree:

Find the split point with the highest information gain *a* (splits data the best), split the current dataset around that value: to the left would be less than *a,* to the right would be greater than *a.* Then recursively do the same steps on the split data in the left and right nodes until you reach a specified depth.

```
Build_Tree(train, max depth, min size)
    root = get_split(train)
    Split_Node(root, max_depth, min_size, 1)
    return root
```

```
Split_Node(node, max depth, min size, depth)
  left[][] = node.b_groups.left
  right[][] = node.b_groups.right
  if left.size == 0 or right.size == 0
    node.left, node.right = To_Terminal(left+right)
    return
```

```
if depth >= max depth
  node.left,node.right = To_Terminal(left), To_Terminal(right)
  return
if left.size <= min size
  node.left = To_Terminal(left)
else
  node.left = Get_Split(left)
  Split_Node(node.left, max depth, min size, depth+1)
if right.size <= min size
  node.right = To_Terminal(right)
else
  node.right = Get_Split(right)
  Split_Node(node.right, max depth, min size, depth+1)
```

Each call to `Split_Node` handles the decision to make for a given node's children, either to continue splitting the data or make a leaf node which serves as a final classifier prediction when used later to predict. If the size of the data in the left or right tables is empty then we must make a leaf node for both of its children because we can't split empty data tables. The next if statement checks to see if the tree is at or past the user specified `max_depth`, if so we need to again make the given node's children into leaf nodes. Finally for each table left and right, we check if their size is greater than the user specified `min_size`, if left or right is smaller, then a leaf node is made for its respective side, if not, split the data again, make a node and continue down the branch.

Predict:

The purpose of `predict` is to predict the classifier value for a given row of data using a decision tree built from complete test data. For a given row of data you are trying to predict a classifier for, drill down the tree going to the left child of the current node if your value is less than, and then to the right, if it's greater than or equal to. Follow these steps until you reach a leaf node, that node is your predicted classifier.

```
Predict(node, row)
  if row[node.b_index] < node.b_value
    if !node.left.leaf
      return predict(node.left, row)
    else
      return node.left
  else
    if !node.right.leaf
      return predict(node.right, row)
    else
      return node.right
```

Given the root of the decision tree and a row of test data, this function follows the tree nodes down to a leaf node which will return a predicted value. If the row's data at the given attribute from `node.index` is less than the Node's split value, and if it's not a leaf node recur to the left node; else go to the right node, if it is a leaf node return the predicted value. This is guaranteed to drill down to a leaf, allowing us to make a prediction.

Get Split:

The purpose of `Get_Split` is to find the best value to split the classifiers in each row of data the best. For each row and column of your dataset pick the value *a*, compare each other row at the specified column to *a.* Splitting the data less than *a* to the left vector, and greater than to the right vector, saving the split value and data with the best Gini score.

```
Get_Split(dataset)
  class_values[] = all unique classifier values
  b_index = inf
  b_value = inf
  b_score = inf
  b_groups[][][]
  for each column i except the last
    for each row j in i
```

```
      groups = test_split(i, j, dataset)
      gini = Gini_Index(groups, class_values)
      if gini < b_score
        b_index = i
        b_value = j
        b_score = gini
        b_groups = groups
  b_split.b_groups = b_groups
  b_split.b_index = b_index
  b_split.b_value = b_value
  return b_split
```

Get_Split greedily goes over all possible splits, using a loop and a nested for loop to traverse each column (except for the classifier column), and each row. At the intersection of each row and column, the dataset is split around that value with `test_split`. Then on the split data, we call `Gini_Index` to calculate how good the split was. With the value generated from `Gini_Index`, we save the best split value and the column it's associated with.

Gini Index:

A Gini score is a measure of the proportion of classifiers in each side of the split; in many cases that value is the last column of data in a given row. Basically we are just comparing proportions of the occurrences of classifier values in each side of the split.

```
Gini_Index(groups, classes)
  n_instances = number of rows in groups
  Gini = 0
  for each side in groups
    size = side.size
    if size == 0
      continue
    score = 0
    for each class in classes
      proportion = 0
      for each row in side
        if row[-1] == class
```

```
        p += 1
      score + (p/size)^2
    Gini + (1- score) * (size/n_instances)
  return Gini
```

Each side, left and right, needs to keep their score values separate, so the first nested for loop iterates over the possible classifications, and within that it keeps a counter `p`, and after each iteration of the classification loop, (p/size)^2 is added to a the ongoing score counter, and after each side left and right, a Gini score is calculated and added up. The result is a score from 0-0.5 of how well something was split.

Evaluate Algorithm:

For this project I abstracted out one more step, implementing a k-fold cross validation algorithm to test the accuracy of different training sizes, tree depths, and minimum table size variables.

`Evaluate_algorithm` works by splitting the original data into n different folds (`n_folds`) and then for each fold we create a tree using that fold's data, then test the accuracy of it on the other folds. So for example we split the data into 3 different folds, we call them folds x, y, and z, for the first loop iteration we use fold y and z as training data and fold x as testing data, then repeat with all of the other pairs.

```
evaluate_algorithm(dataset, n_folds, max_depth, min_size)
  folds[][][] = c_v_split(dataset, n_folds)
  scores[]
  for each fold in folds
    train_set[][]
    test_set[][]
    insert all folds except current fold into train_set
    for each row in fold
      row_copy[] = row
      row_copy[-1].delete
```

```
      test_set + row_copy
    predicted[] = decision_tree(train_set,test_set,max_depth,min_size)
    actual[]
    for each row in fold
      actual + row[-1]
    accuracy = accuracy_metric(actual, predicted)
    scores + accuracy
  return scores
```

`c_v_split` is the algorithm that splits the dataset into n different folds, and `accuracy_metric` compares the predicted classifier values to the actual values and returns a percentage accuracy value.

Runtime Analysis:

Just Tree:

- O(r * c * log(m))
    - r = rows in data sheet
    - c = attributes in data
    - m = max_depth
- r *c comes from running the greedy `get_split` algorithm on each node, then the log(m) comes from the average height of the tree.

+ Evaluation:

- O(n * r * c * log(m))
    - n = number of folds
- By evaluating the algorithm we get a whole new layer of commands, this is because for every fold we create, we need to build a tree.

Nolan Brady

Citations:

- History: https://www.explorium.ai/blog/the-complete-guide-to-decision-trees/
- Tutorial:
    - https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/
    - https://www.youtube.com/watch?v=RmajweUFKvM&t=1013s